A SOURCE CODE ANALYZER TO PREDICT COMPILATION TIME FOR AVIONICS SOFTWARE USING SOFTWARE SCIENCE MEASURES

THESIS

Volume I Main Report

AFIT/GCS/ENG/88D-7 Eric R. Goepper

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

0

AFIT/GCS/ENG/88D-7

A SOURCE CODE ANALYZER TO PREDICT COMPILATION TIME FOR AVIONICS SOFTWARE USING SOFTWARE SCIENCE MEASURES

THESIS
Volume I Main Report

AFIT/GCS/ENG/88D-7 Eric R. Goepper
Captain, USAF

AFIT/GCS/ENG/88D-7

A SOURCE CODE ANALYZER TO PREDICT COMPILATION TIME FOR AVIONICS SOFTWARE USING SOFTWARE SCIENCE MEASURES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements of the Degree of
Master of Science

Eric R. Goepper, B.S.
Captain, USAF

December 1988

Table of Contents

	Page
Preface	ii
List of Figures	iv
List of Tables	v
Abstract	vi
I. Introduction and Problem Statement	1
II. Software Metrics and Software Science Measures	8
III. Design and Construction of the SCA	22
IV. SCA Implementation Details	35
V. Testing and Results	45
VI. Conclusions and Recommendations	70
Appendix A1: Analysis of Original Compile Time Model	79
Appendix A2: Linear Regression	89
Appendix A3: Analysis of Recalibrated Compile Time Model	109
Appendix B: User's Manual for the SCA	124
Appendix C: Sample SCA Output	129
Bibliography	139
Vita	142

Preface

This thesis used existing software tools, metrics theory, and a prototype model for compile time to build a computer program which is simple to use, easily ported, and works as advertised. The program can be a useful tool for software metrics researchers as well. What's more, there are no flies in the ointment. There isn't a command set to learn nor a list of subtle run-time nuances of which to be aware. If you follow the installation instructions in the User's Manual, I believe the program will work reliably in your environment without requiring any programming changes. The program should be portable to any mainframe computer system which uses the popular UNIX operating system, offers a C language compiler, and supports the compiler generating tools LEX and YACC (they are normally bundled with UNIX by the vendor). The number of installations meeting these requirements is already quite large and growing respectably.

I would like to thank my thesis advisor, Major Jim Howatt of the Air Force Institute of Technology, for his suggestions regarding the architecture of the program and for his insights involving software metrics and compiler theory. I would also like to thank two committee members who are also with the Institute. Capt Wade Shaw assisted with the statistical analysis and helped interpret the data and the behavior of the timing model. Capt Dave Umphress provided an insightful review of the draft manuscript which helped improve the overall quality of the work.

Eric R. Goepper

List of Figures

Figure Page

1. Design for SCA Development 25
2. SCA Construction Plan 27
3. Contents of Ada.y 30
4. Contents of Y.tab.c 31
5. Identifier Table Entry 36
6. Table Management Functions 37
7. Sample Output of Delimiter Counts 40
8. Shell File For Testing CAMP Software 54
9. Sample Records in the Measures File 55
10. Predicted and Actual Compile Time 57
11. Data Point Distribution 61
12. Prediction Error Versus File Length (N) 63
13. Actual Compile Time Versus File Length (N) 65
14. Comparison of Predictive Models 67
15. Recalibrated Model Statistics 68

List of Tables

Tables Page

1. Software Science Measures 10
2. Halstead's Equations 11
3. Constants 20
4. Sample Identifier Table 41
5. Sample Compile Time Predictions 44
6. Three Data Item Definitions 56
7. Prediction Error Statistics 59

AFIT/GCS/ENG/88D-7

Abstract

This thesis describes the construction of an Ada source code analyzer (SCA) which produces values for the Software Science measures n_1, n_2, N_1, and N_2. The measures are used to evaluate a mathematical model designed to predict the compile time of Ada modules. The primary goal of this effort was to provide a software tool to metrics researchers which could automatically compute Software Science measures for Ada modules. A secondary goal was to produce a convenient method for Ada compiler researchers to predict the amount of time consumed during compilation of given avionics software modules.

As the SCA was built, we incorporated the rules of a new Ada token counting strategy designed to yield meaningful results for entire Ada programs, not just executable code. Once satisfied the SCA implemented the rules correctly and produced accurate counts for the Software Science measures, we added the compile time model to the SCA.

To test the validity of the compile time model, over 200 modules were selected at random from among the Common Ada Missile Packages (CAMP) software library. For each module chosen, both the compile time as predicted by the SCA and the actual compile time using the Verdix Ada compiler were recorded. Finally, the

prediction error values (predicted compile time minus actual compile time) were recorded and analyzed. (KR) ←

For the test environment we used, in 95% of the test cases the SCA initially overestimated compile time with an average prediction error of 4.35 seconds. Since the average actual compile time was only 3.88 seconds, this average error figure was unacceptable. In addition, the magnitude of the prediction error increased disproportionately as the size of the module increased. These results led us to recalibrate the model's parameters. When the recalibrated model was tested, the average error fell to -.25 seconds. This value was much more respectable in view of the actual compile time average. Moreover, the curve of the predicted compile time values now fit the curve of the actual values nicely.

A SOURCE CODE ANALYZER TO PREDICT COMPILATION TIME FOR AVIONICS SOFTWARE USING SOFTWARE SCIENCE MEASURES

I. Introduction and Problem Statement

Origin of the Problem

Various aspects of the software development process affect the quality of the software product. The quality of the software requirements, specifications, and code directly affects the ultimate value of the software produced. One goal of software engineering is to produce the "best" software product in the most efficient and cost-effective manner. Intuitively, we know that how we put the software together must have a great deal to do with how well it meets requirements, how hard it is to maintain, how many errors it has, and so on. One of the most important determinants in this process is the compilation process.

One of the difficulties faced by military software development agencies is the problem of choosing the best compiler for a particular computer software development environment. Generally these agencies are beset with the following two questions:

1. Since Ada is the mandatory development language, how should the most efficient Ada compiler be chosen for an avionics software development environment?

2. Is it possible to predict the performance of validated Ada compilers with respect to compilation time for avionics software?

The validation process mandated by the Department of Defense (DOD) for Ada compilers does not guarantee the efficiency of the compiler; in fact, compilation resources such as compile time and object code size are not criteria in the validation process. For example, a situation could occur in which one validated compiler produces a 50 kilobyte (K) object code module in 180 seconds, while a different validated compiler outputs a 70 K object module in 150 seconds using the same source program as input. The dilemma of the development agency is clear: given these widely varying parameters, which one of these two compilers should be chosen for a particular development environment?

At the DOD Ada Validation Facility located at Wright-Patterson AFB, Ohio, the need exists to measure the performance of validated Ada compilers in the context of avionics software development. Managers at the Facility are concerned with measuring and predicting the efficiency of Ada compilers for avionics systems. For example, they would like to know whether or not a new compiler is faster than established compilers. They would also like to identify the specific language constructs used in Ada avionics software which require relatively more overhead to compile.

The specific question which developers face is: in terms of compilation time, how can different Ada compilers be evaluated relative to one another so the best one can be chosen for a particular software development task? The answer might involve complexity metrics.

In its broadest sense the term "software metrics" refers to the branch of software engineering which is concerned with objectively and rigorously quantifying those aspects of the software development process which affect software quality. In general, researchers in the field of software metrics attempt to answer the following two questions: Given an observed behavior of the software product (e.g., error-prone, hard to debug), what is it about the way the source code was developed that gives rise to this behavior? How can we measure the phenomenon? Researchers focus on the ways that software metrics can be used to *predict* program behavior based on an analysis of the source code text.

Since the length of time required by compilation is thought to be an increasing function of the complexity of the software, researchers believe complexity metrics can be used to predict compilation time for a given set of source code programs (Howatt, 1988). In other words, if metrics can be used to measure the complexity of software modules, it should be possible to use them to help predict the amount of time required to compile a software module.

Problem Statement and Goals

Currently the DOD Ada Validation Facility does not have an automated software tool to predict the compilation time of avionics source code modules. The goal of this thesis is to produce such a tool, called a source code analyzer (SCA), which correctly implements a pre-defined counting strategy for Software

Science measures, and automatically predicts compile times for Ada source code modules as well.

Scope

The study is limited to the specific problem of producing a SCA which implements a model to predict compile times for syntactically correct software coded in Ada. The only measurement of the compilation process is compile time. We define compile time as the Central Processing Unit (CPU) time which elapses between the start of compilation and the moment when the resulting object code file has been successfully produced. Other aspects of the compilation process (e.g., number of external references, object code size) are not addressed in this work.

The SCA can be used only as a tool for the prediction of compilation time for Ada software modules. The data produced by the SCA can help researchers evaluate the compilation efficiency of Ada compilers. However, the SCA does not directly offer information concerning other characteristics or performance parameters of avionics software modules or Ada compilers. For example, neither a prediction of the object code size of modules output by a compiler nor a prediction of possible run-time errors is available from the SCA.

General Approach

To construct the SCA, an Ada source code parser was generated, and a recently published Ada token counting strategy together with a prototype compile time model were implemented in the program. As a test of the validity of the timing model, over 200 software modules (files) were input to the SCA, and the predicted compile time was compared to the actual compile time for each module.

The compilation timing model is borrowed from (Miller, 1986). Miller derived this model using Software Science measures computed from Ada source code. These measures had been *manually* generated using the same token counting strategy as implemented in the SCA. This counting strategy was developed at the Air Force Institute of Technology and supports token counting for the full Ada language (Miller and others, 1987).

Assumptions

The key assumption we made is that Miller's timing model offers a reasonable theoretical foundation on which to build. We fully anticipated the necessity for fine tuning or recalibrating the model. In the planning stage we accepted the risk that the ability of the SCA to initially predict an accurate range of compile times would depend for the most part on the validity of Miller's basic model, and only indirectly on the correctness of the SCA. Although there was reasonable evidence that the model was valid (Miller, 1986), the model had not been empirically validated before we implemented it in the SCA.

Sequence of Presentation

Chapter II discusses the applicable topics of Software Science measures, presents the Ada token counting strategy developed at AFIT, and overviews the compilation timing model.

Chapter III covers the design and construction of the SCA. Since the SCA was generated using existing software tools, background material for these tools is reviewed at the beginning of Chapter III.

Chapter IV discusses in detail the implementation of the routines responsible for data structure management and token counting support in the SCA. In addition, the Software Science measures tabulation and the timing model computations are covered.

A discussion of the methods employed for correctness testing of the SCA begins Chapter V. Considering the token counting strategy as "design specifications", a few caveats and deviations from the strategy are discussed next. The results of testing the SCA compile time predictions and the recalibration of the timing model are presented and analyzed in Chapter V.

Conclusions regarding the correctness, validity, and the applicability of the SCA are discussed in Chapter VI. Specific conclusions regarding compile time models for Ada source code are listed too. Recommendations for enhancement of the SCA and ideas for further research employing the SCA are also presented in Chapter VI.

II. Software Metrics and Software Science Measures

The software industry and academia have investigated ways to determine and measure the various factors which influence software quality. One of the areas which appears to be highly active is software complexity metrics. By definition, a complexity metric is a single number or small set of numbers derived from an analysis of the source code. These numbers may represent a measure of the program size, flow of control, flow of data, structure of the data, or the degree to which the code was structured. Using such metrics, for example, one could assert that program "A" is more complex than program "B", in at least certain respects. (The terms "metric" and "measure" are used synonymously in the vocabulary of software metrics.)

Software metrics are usually derived from an analysis of programming language source code. Most of these metrics have a unique approach or technique to analyze the code. Usually this takes the form of an algorithm and one or more equations associated with that algorithm. Characteristics of the code are analyzed using the algorithm and equations. Based on calculations from the equations, an output value or metric is produced. Notice that the term "metric" is used to mean both the particular technique of measurement as well as the numerical value produced by that measurement.

For example, the simplest algorithm would be a count of the lines of source code. The algorithm in this case would be: sum the number of lines of code. This simple metric is still studied empirically to determine its merit as a measure of software complexity. The other early algorithms, and the ones upon which much of current research in complexity metrics is based, involved counting the lexical elements or counting the number of possible control (execution) paths in a program.

There are many different types of complexity metrics in the literature, each metric dealing with one or more aspects of software complexity. There has not, however, been a great deal of empirical evidence offered in support of these metrics. Most of the researchers have relied on the "intuitive appeal" of their metric, which they usually claim is inherent in the design and construction of their metric. Indeed, the majority of metrics have not been empirically proven valid (Howatt, 1988). Part of the problem has been the lack of rigor and precision in the definition of the metrics and their components and/or equations. Many intuitively appealing metrics lack validation because their definitions or methodology are vague and imprecise.

One attempt to help mitigate this problem was an effort to characterize the fundamental concepts and lexicon of control flow metrics more rigorously (Howatt, 1985). In addition, recently proposed metrics (e.g., Measure Based On Weights) have been designed to permit a more credible approach to empirical testing (Jayaprakash and others, 1987). But despite these advances, validated metrics are the exception rather than the rule.

Software Science Measures

In the early 1970s Maurice Halstead, while working at Purdue University, originated a body of metrics theory which has become known as the Software Science Theory of metrics (Halstead, 1977). His work developed as an outgrowth of research involving the analysis of software complexity using mathematical algorithms (Halstead, 1977). Software Science *measures* are counts of operators and operands in a software module. The measures are defined in Table 1.

By definition, a *token* is an operator or an operand. An *operator* is any function, symbol, or group of symbols in the code that produces an action (e.g.,+, your_function(), sqrt(x)). An *operand* is any type of constant or variable (e.g., my_count, pi, 3.14159) in the source code. An operator normally acts upon, modifies, or in some way makes use of an operand. Operands are generally user-defined; operators are often part of the language itself. An exception to this rule are user-defined subprograms which, when invoked in the code, are considered operators.

Table 1. Software Science Measures (Halstead, 1977)

n1	=	number of unique operators
n2	=	number of unique operands
N1	=	total occurrences of all operators
N2	=	total occurrences of all operands

Halstead then developed a set of software metrics which are based on the Software Science measures. These metrics are defined in Table 2. Some of these metrics are central to Miller's theoretical development of the compilation timing model.

Table 2. Halstead's Equations (Halstead, 1977)

$$\text{Vocabulary} = n = n1 + n2$$

$$\text{Length} = N = N1 + N2$$

$$\text{Est. Length} = N\hat{} = (n1 * \log2(n1)) + (n2 * \log2(n2))$$

$$\text{Volume} = V = N * \log2(n)$$

$$\text{Est. Volume} = V\hat{} = N\hat{} * \log2(n)$$

$$\text{Potential Volume} = V* = (2 + n2*) * \log2(2+ n2*)$$

$$\text{Level of Implementation} = L = V* / V$$

To compute the metrics of Table 2, the source code of a module is scanned and analyzed by a manual or automated method. As the code is scanned, all tokens encountered are counted according to a pre-defined set of rules called a *token counting strategy*. Since a token can be an operator *or* an operand, the strategy must arbitrate between instances of operators and operands depending on the current context. Eventually this counting process produces the measures n_1, n_2, N_1, and N_2 of Table 1. These values are then substituted into the appropriate equation from Table 2, and the value for a particular metric of interest is computed. (Note: the term n_2^* in Table 2 is a theoretical measure which requires special handling. For an explanation of n_2^*, see Miller, 1986 or Halstead, 1977.)

Evaluation of Software Science Metrics. The literature is full of criticism of the Software Science theory and methodology (Levitin, 1986). Although considered by many researchers to be useful, Software Science measures suffer from many deficiencies. A few of the allegations against these measures are listed here.

1. They do not measure control flow complexity (Ramamurthy and Melton, 1986:309).

2. Their credibility suffers from a lack of empirical and analytical evidence to validate them (Howatt, 1985:40).

3. The counting process does not take into account non-executable statements (Levitin, 1986:314), despite the fact that it has been shown that declarations and other non-executable statements contribute to complexity.

4. It is difficult to separate tokens into disjoint operator and operand sets (Levitin, 1986:317).

5. The equation for volume is basically flawed in the sense that it is not additive (Levitin, 1986:317).

6. The counting rules cannot be applied consistently across several languages (Mehndiratta, 1987:370).

7. The measures do not reflect modularity (Van Verth, 1987:252).

Although some experiments have produced empirical evidence in support of Software Science measures, researchers agree that for the most part they remain un-validated (Levitin, 1986). Unquestionably these metrics fail to embrace all the factors which contribute to program complexity (Van Verth, 1987).

Most of the criticism of Software Science measures is levied by researchers who are concerned primarily with the psychological complexity of software modules. In this effort we are not interested in the psychological aspects of complexity. Instead, the consumption of resources (specifically time) during compilation is our focus. Since a compiler processes tokens as its basic unit of work, it makes sense to apply Software Science measures in our research.

The Ada Token Counting Strategy

In "A Software Science Counting Strategy for the Full Ada Language" the authors present a new token counting strategy for the full Ada programming language (Miller and others, 1987). This new counting strategy extends Halstead's original method of counting tokens and adapts it to the full Ada language. The major tenet of the new rules is to categorize the syntactical language constructs according to the amount of work or *overhead* they cause the compiler. For example, although the Ada construct "for ... in ... loop ... end loop" contains 5 tokens, the compiler handles them as one semantic structure rather than as a string of 5 independent reserved words. According to the token counting strategy, these words would be counted as one multi-token operator. In this way, the actual parsing strategy of the compiler is captured more accurately.

The new token counting strategy features context-based rules to help differentiate between operators and operands (Maness, 1986; Miller, 1986). Additionally, Miller and others extended Halstead's original token counting strategy to include the count of non-executable source code (Maness, 1986; Miller, 1986). In short, the strategy counts a larger set of Ada source statements and uses concise, unambiguous, context-based rules for differentiating operators from operands.

The counting rules are listed below. *These rules constitute the baseline definition of the token counting strategy currently implemented in the SCA.* For explanations of the development and interpretation of the rules see Miller and others, 1987 or Maness, 1986. For examples relating to the use of some of these rules, inspect the sample SCA output listings in Appendix C. These listings provide the reader with numerous examples illustrating the token counting strategy and its rules.

1. All entities, except comments, in a module are considered.

2. Variables, constants, and literals are counted as operands. Local variables with the same name in different procedures/functions are counted as unique operands. Global variables used in different procedure/functions are counted as multiple occurrences of the same operand.

3. The following pairs of tokens are counted as multi-token single operators:

FOR USE	SELECT END SELECT
DO END	DECLARE BEGIN END

OR ELSE	LIMITED PRIVATE
BODY IS	FUNCTION RETURN
ARRAY OF	RECORD END RECORD
AND THEN	FOR IN LOOP END LOOP
BEGIN END	WHILE LOOP END LOOP
SUBTYPE IS	CASE IS WHEN END CASE
ELSIF THEN	LOOP END LOOP
IF THEN END IF	EXCEPTION WHEN
GOTO	

4. The following tokens or pairs of tokens are counted as single operators subject to the accompanying conditions:

+ is counted as either a unary + or a binary + depending on its function. A unary + is not counted when it is part of a numeric constant like +2.15.

- is counted as either a unary - or a binary - depending on its function. A unary - is not counted when it is part of a numeric constant like -22.5.

( ) is counted as either (1) an expression grouping operator as in (x+y) / z , (2) an invocation operator, as in xx := sqrt(a), (3) a declaration operator, as in PROCEDURE xx(a : in REAL), (4) a subscript operator, as in x := i(j), (5) a dimensioning operator, as in k : array(1..5) of REAL, (6) an aggregate operator, as in x : f_type := (OTHERS => ' '), (7) an enumeration operator, as in type color is (red, green, blue), or (8) a type conversion operator, as in int := integer(real_variable).

' (apostrophe) is counted as either (1) an attribute operator, or (2) an aggregate operator. A pair of apostrophes used in character constants, such as 'x' is counted as a single operator.

IN is counted as either (1) a mode operator, or (2) a membership test operator.

OR is counted as either (1) a boolean operator, or (2) a alternative operator in SELECT statements.

NULL is counted as either (1) an operator if it appears in executable code, or (2) an operand when used as a constant.

PRIVATE and SEPARATE are counted as either declaration operators or as detail operators.

5. The following tokens are counted as single operators if they are not used in rules 3 and 4:

:   >   <   ,   &   ;   *   /   .   <=   ..
=
>=   /=   =>   **   < >   # #   " "   :=   <<>>   ¦   IS   AT
ABS REM END XOR AND MOD USE NEW ALL NOT OUT ELSE
TYPE TASK EXIT WHEN RANGE RAISE ABORT OTHERS DELAY
DELTA WITH ENTRY DIGITS GO TO GENERIC ACCEPT RETURN
ACCESS PRAGMA REVERSE EXCEPTION TERMINATE CONSTANT
PACKAGE RENAMES PROCEDURE

6. Procedure and function calls are counted as operators.

7. A type indicator is counted as either (1) an operand in its own declaration statement, or (2) an operator if it types a variable, function, or subtype.

8. "Package/Procedure/Function Is New" is called a generic instantiation operator and is counted as one unique operator.

The Compile Time Model

The compile time model implemented in the SCA is from the thesis "Application of Halstead's Timing Model to Predict the Compilation Time of Ada Compilers" (Miller, 1986). In that work, Ada source code modules were manually counted according to the token counting strategy just presented, and the Software Science measures produced were used to derive the model. For a detailed account of the derivation of the model, see Miller 1986, especially pages 32, 35, and 67.

A very brief overview of the derivation is provided now. The theoretical basis of the timing model begins with the last equation in Table 2:

$$\text{Human Programming Time} = T = V^2 / (S * V^*) \quad (1)$$

V and V^* represent volume and potential volume, respectively. S represents the "discrimination rate" of a typical human programmer. Miller's working hypothesis asserted that the amount of time, T, required for a human to program a software module should be related in some meaningful way to the amount of time required to compile the same module on a computer (Miller, 1986).

Miller showed that Equation 1 could be expressed in parametric form as:

$$T = K * V^a * (V^*)^b \tag{2}$$

In this equation T represents the predicted compile time of the module, and K plays the role of an environment-specific constant; K roughly corresponds to the inverse of S in Equation 1. The exponents a and b are the *parameters* of Equation 2, where $a = 2$ and $b = -1$.

Miller accumulated a data base of values for the Software Science measures n_1, n_2, N_1, N_2, N, n, and n_2^* by manually counting Ada source code modules. The new measure here, n_2^*, represents the number of input/output parameters in a module. n_2^* is important because, as a reference to Table 2 will reveal, it is used in the computation of V^*. Seeking a model for compile time, Miller used the data and the technique of linear regression to derive and verify *new* values for the parameters a and b in Equation 2 (Miller, 1986, 67). These values turned out to be fractions, differing from one another by a full order of magnitude as we see in the next equation.

$$T = K_i * V^{(.4839)} * (V^*)^{(.0745)} \qquad (3)$$

In Equation 3, K_i is further defined as one of four environment-specific constants for i = 1,2,3 and 4. Both V and V^* remain the same. Obviously, the exponents a = .4839 and b = .0745 are the parameters for the compile time model. These parameters are not only environment-specific, but also Ada compiler specific. For purposes of this study we shall initially accept these exponents at face value; we're not so much concerned here with the methods used to obtain them as we are interested in putting them to use in the SCA.

At this point, it looks as if we can implement Equation 3 in the SCA without difficulty. But recall that to compute V^* we need a value for n_2^*. The SCA does not compute this particular measure, however. Fortunately, we can make use of the metric for estimated level, $L^\wedge$, in Table 2 as a legitimate substitute for potential volume V^* in Equation 3. This substitution is motivated by both Miller's work and Halstead's original writings (Miller, 1986, 35). In fact, Miller actually offered an "estimated" version of Equation 3 which employs this substitution (Miller, 1986, 55). The estimated version of Equation 3, shown in Equation 4 below, also substitutes volume V with the estimated volume $V^\wedge$. Equation 4 can be used as an approximation to Equation 3 when the latter cannot be easily computed.

$$T = K_i * (V^\wedge)^a * (L^\wedge)^b \qquad (4)$$

Since the SCA produces all the measures required to compute volume V, there is no need for the compile time model to use $V^{\hat{}}$ instead of V itself. Therefore, the basic expression for compile time implemented in the SCA uses V and $L^{\hat{}}$ as follows:

$$T = K_i * V^{.4839} * (L^{\hat{}})^{.0745} \qquad (5)$$

The reader should be aware of the following important fact: the values of a and b (which Miller found to be .4839 and .0745 respectively), *were originally derived for Equation 2, not* ***Equation*** *4*. The SCA "borrows" the parameters derived for one expression of the compile time model and uses them as parameters in a similar, yet different, expression for the same model. Although this approach was risky, any merits or demerits would be eventually reflected in the test results. We were reasonably assured that the substitution of V^* by $L^{\hat{}}$ couldn't impact the magnitude of the predicted compile time a great deal since the exponent of the V^* term, .0745, was small relative to the exponent .4839 of the unadulterated term V.

To compute Equation 5 in the SCA, we substitute the definitions for V and $L^{\hat{}}$ from Table 2 appropriately in Equation 5. Since V and $L^{\hat{}}$ are both defined in terms of the Software Science measures n_1, n_2, N_1, N_2, N, and n, the model for predicted compile time T can also be expressed as shown below in Equation 6. This became the final expression for the compile time model before it was coded into the SCA.

$$T = K_i\ (N * \log_2(n))^{.4839} * ((2 * n_2) / (n_1 * N_2))^{.0745} \qquad (6)$$

As we know, the value of K_i substituted into Equation 6 depends on the particular hardware/software environment in which the source code module will be compiled. Miller derived four specific values for K_i corresponding to four different compilers in four unique computer environments. These values are listed in Table 3 below (Miller, 1986).

Table 3. Constants

K1 =	.3746	(ASC UNIX)
K2 =	.2140	(ISL VMS)
K3 =	.1924	(CSC VMS)
K4 =	.3369	(DG AOS)

The acronyms ASC, ISL, CSC, and DG stand for specific computer systems operated by the Air Force Institute of Technology (AFIT) or the DOD Ada Validation Facility. The remaining acronyms refer to the particular operating system deployed on that system. The ASC is the primary system of interest in this work. The Academic Support Computer (ASC) is a Digital Equipment Corporation VAX-11/785 running the Berkeley 4.3 UNIX operating system. It features 8 megabytes of main memory and over 1300 megabytes of disk storage. All testing of the compile time model implemented in the SCA took place on the ASC.

We can view the constant K_i as a performance index for compiler efficiency; that is, K_i represents the processing or translating rate of a particular Ada compiler in the context of a particular hardware/software environment (Miller, 1986). Since the SCA implements Equation 6 as the compile time model, the value of T in Equation 6 is computed for each of the values of K_i shown in Table 3. These four compile time predictions are also printed by the SCA for the user.

However, the compile time predictions of the SCA were evaluated for the ASC environment *only*. Since this environment requires the use of K_1=.3746, all predicted and actual compile time values used for *testing* use this value of K_1 in Equation 6. The SCA was not tested in the other three computer environments.

III. Design and Construction of the SCA

By way of quick review, the plan to develop the SCA was organized as follows:

1. Scan the literature and choose the compilation timing model to implement in the SCA.

2. Build the SCA, implementing and testing both the token counting strategy and the timing model in the process.

3. Apply the SCA to over 200 Ada source code files on the ASC to examine the validity of the timing model and the behavior of the SCA in that environment. Recalibrate the model if necessary.

Since the timing model was discussed at length in the last chapter, only the design and construction of the SCA will be presented in this chapter after a brief introduction to two software tools which were key to the development of the SCA. The correctness testing and behavior of the SCA will be covered in Chapter IV.

The LEX and YACC Tools

The SCA was developed by combining some original software with existing, public-domain software. The approach involved the use of the YACC and LEX software tools which run under the UNIX operating system. LEX is a program which produces a lexical analyzer for the source code of a specific programming language. Called a *scanner generator*, the LEX program accepts an input file which must contain regular expressions defining the lexical elements (tokens) of the language. For a further discussion of lexical analysis and regular expressions, see (Fischer and Leblanc, 1988).

LEX takes a file of regular expressions defining the lexical constructs of the language and outputs a deterministic finite automaton (FA) recognizer for any pattern of tokens which can be generated by those regular expressions (Kernighan and Pike, 1984). Another way to express what LEX does is to say that the LEX program produces tables (or tabular representations) of the FA's transition diagrams as well as routines which use these tables to recognize the tokens of the language (Kernighan and Pike, 1984). The output file is a source code scanner which accepts "raw" source code as input and consecutively passes valid tokens to a parser. The scanner can also be designed to reject illegal tokens and print an error message.

YACC (Yet Another Compiler Compiler) is a program which produces a parser for the source code of a specific language. Called a *parser generator*, the YACC program accepts an input file which contains the syntax rules of the language expressed as a context-free grammar (CFG). For the SCA, the CFG we used is based on the CFG found in Appendix E of the Ada Language Reference Manual (Mil-Std-1815A). For a further discussion of parsing, parser program, or CFG, see (Fischer and Leblanc, 1988).

The output of YACC is a LALR(1) type source code parser. The "LA" and the "(1)" denote the capability of the parser to "look ahead" one token at a time in order to best determine the correct parse for the sequence of tokens. The "LR" indicates that the parse will produce a right-most parse (bottom-up), as opposed to a left-most parse (Fischer and LeBlanc, 1988). Importantly, YACC offers the user the ability to put C language source code routines into its input file. These routines are later copied by YACC verbatim to the generated program. The copied routines are usually the semantic actions taken when productions are recognized (reduced). These C routines could be anything the designer wants the parser to do upon recognition of a valid syntactic construct (Fischer and Leblanc, 1988).

SCA Design Strategy

Because of time constraints on producing the SCA, a scanner/parser had to be built quickly. Therefore, we chose to use the scanner/parser generators LEX and YACC in the interest of time. Our goal, however, was not simply to build a scanner/parser. Our primary goal was to build a program to tabulate Software Science measures and predict compile times for Ada source code modules. So we used these available tools to allow the majority of the development time to be spent focusing on the problem of integrating the token counting strategy *within* the parser. In short, since the implementation of the rules was paramount, it seemed reasonable to use LEX and YACC to our advantage.

Figure 1, Design for SCA Development, shows the manner in which the two software tools LEX and YACC were used in the development of the SCA.

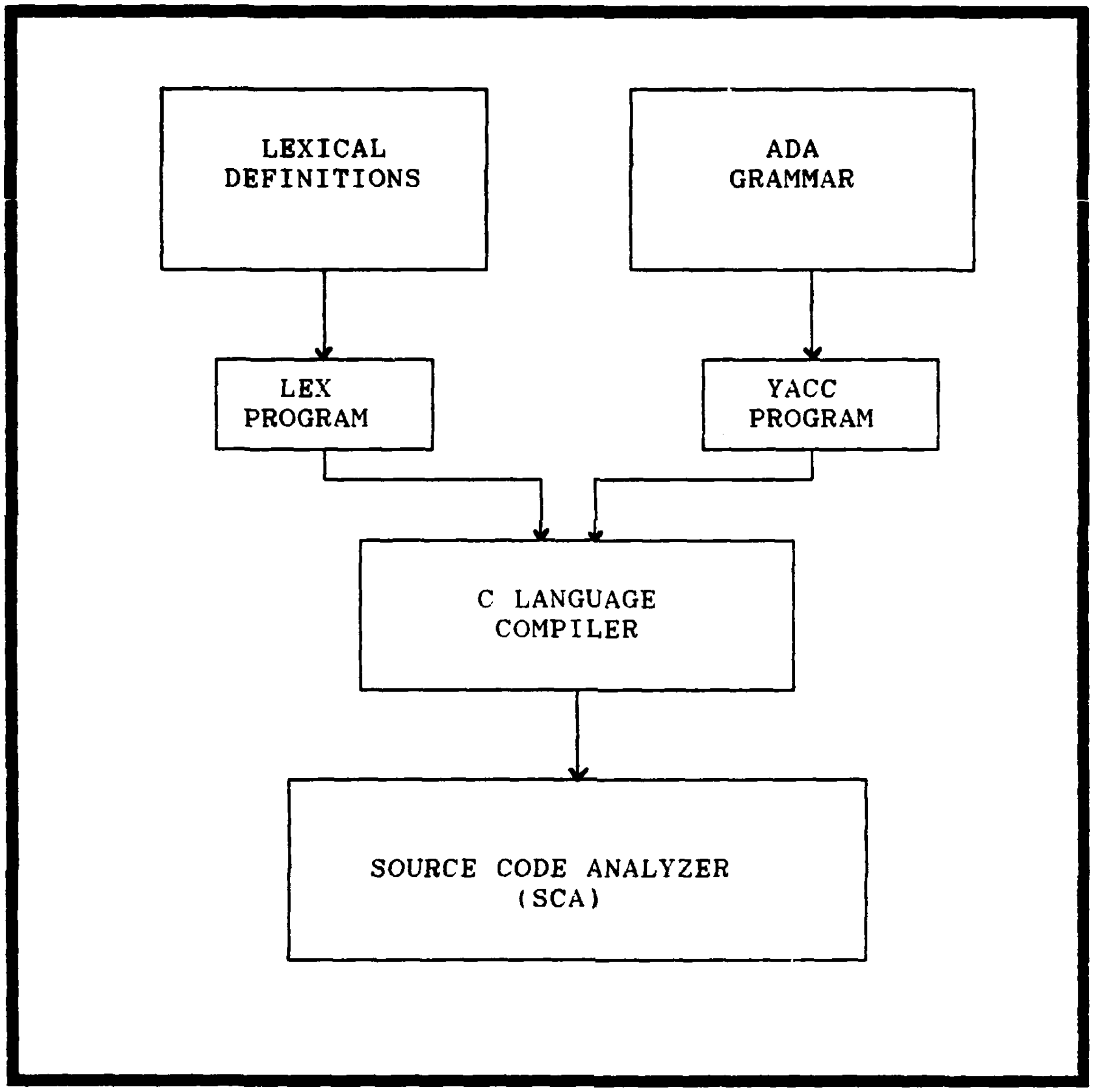

Figure 1. Design for SCA Development (Howatt, 1988)

The input files to LEX and YACC which contain the lexical definitions and Ada grammar respectively, are shown at the top in

Figure 1. Not shown, however, are the output files from LEX and YACC which contain the generated scanner and parser. The fact that the scanner and parser must be compiled is represented by the arrows into the C compiler. The output files from LEX and YACC are discussed in the next section.

SCA Construction

Figure 2, SCA Construction Plan, provides a file-level perspective of the mechanics of the SCA construction. The input files to LEX and YACC, Ada.l and Ada.y, are of interest here first. Ada.l contains the lexical definitions for LEX, and Ada.y contains the Ada grammar for YACC. Both files were obtained from the Ada Software Repository. They were originally built by Herman Fischer of Litton Data Systems in 1984. The documentation header for the Ada.y indicates that the file contains a "LALR(1) grammar for ANSI Ada" that has been "adapted for YACC inputs". Original source listings for Ada.l and Ada.y are in Volume II of this thesis, available through the Department of Electrical and Computer Engineering at AFIT.

During development of the SCA, some of the regular expressions in Ada.l and quite a number of the CFGs in Ada.y were changed in order to implement the counting strategy. However, only one change was made which rendered the grammar inconsistent with the Ada LRM (for that one case, see Caveats and Deviations, Chapter V). Listings of the modified versions of Ada.l and

Ada.y, which together constitute the "source code" for the SCA, are also in Volume II of this work.

Notice that the file lex.yy.c shown in Figure 2 "includes" the file "y.tab.h". Y.tab.h contains the C language statements to associate a unique code with each reserved word in the Ada

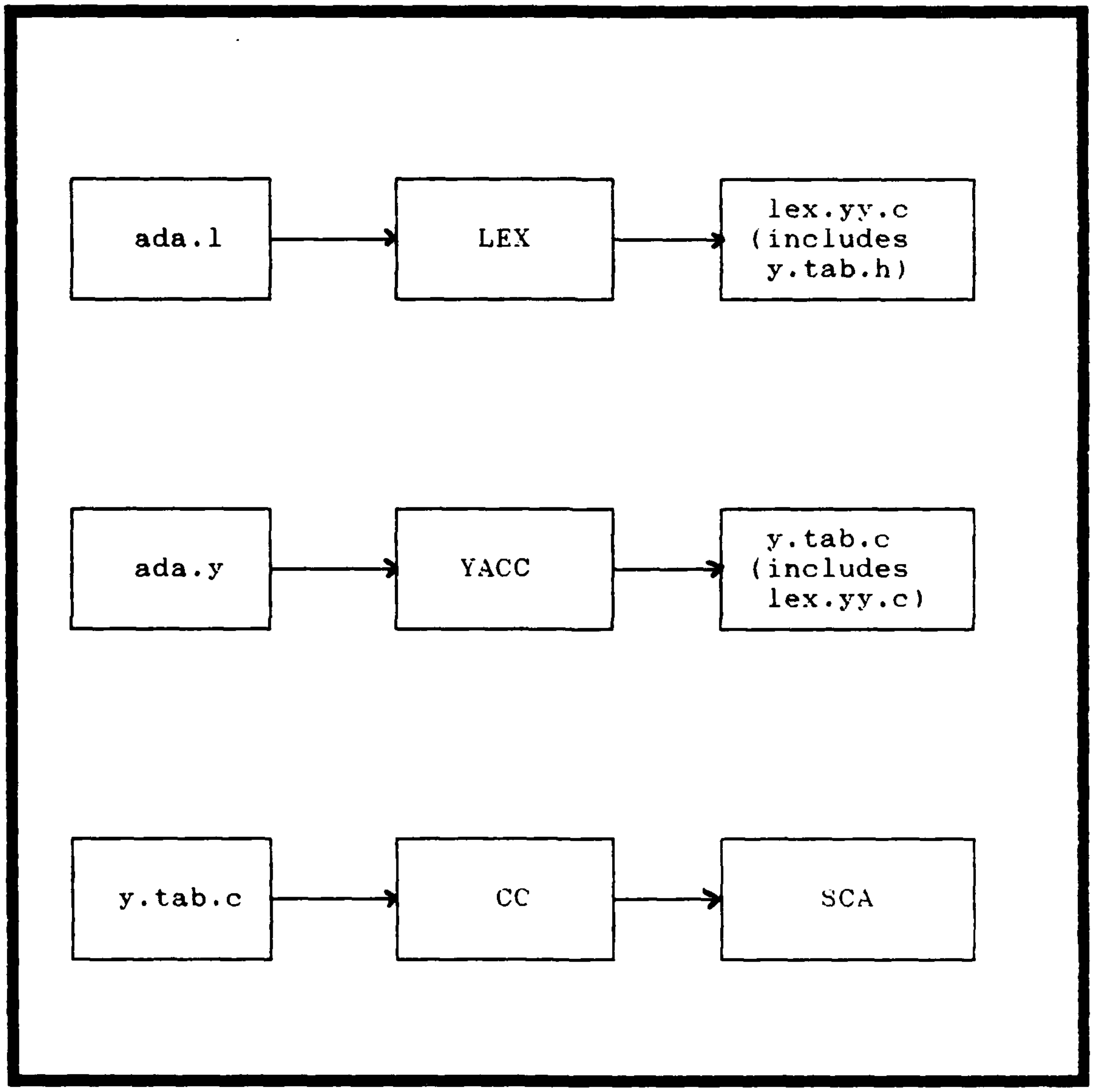

Figure 2. SCA Construction Plan

language. This is done because the parser can work more easily with a numerical code than with the actual token string.

Implementing the Counting Strategy in the SCA. The ability to invoke semantic actions (or any other tasks) at virtually any point in the parsing or token recognition process was key to building the SCA. In fact, inserting C language statements in the parser's CFG to implement the token counting strategy was the heart of the development. Specifically, C statements and subroutine calls were integrated into Ada.y at judiciously chosen points in the source code text of the CFG where a construct of interest was recognized by the parser.

The type and purpose of the token counting code to be integrated at a given point in the text of a CFG depended upon the particular construct or token which was *recognized* at that point in the CFG. In addition, the code was intentionally placed at a point in the text such that whenever the parser recognized that particular construct during parsing, the intended statement or routine would be immediately executed.

There were essentially two types of tasks performed by these routines. The first simply involved incrementing a counter. Since this task was relatively simple, a single statement rather than a subroutine call was often used. The second type of task was more complex. It involved storing the name or identifying phrase belonging to the current syntactic construct in a suitable data structure. We called this data structure the "Identifier Table", and implemented it logically as a singly linked-list. This particular implementation was chosen because the amount of storage required could never be determined in advance (i.e, an upper limit for the number of tokens in the source modules could not be set and we considered sequential searching techniques to be adequate for the SCA). It was important that not only the name of the identifier, but also other pertinent information about the identifier be stored as a single entry (record) in the Identifier Table.

Modifying the YACC input file, Ada.y, required the bulk of the programming effort. Often there was a significant amount of overhead just to determine the correct information pertaining to an identifier and to get the data properly stored in the Identifier Table. A sketch of the organization and contents of Ada.y are displayed in Figure 3. The file contains the following components: the Ada grammar; code for the token counting strategy; the Software Science measures tabulation routines; and Miller's compilation timing model. Notice that Ada.y uses the C "include" directive for the file lex.yy.c. This is done so that the parser can call the scanner to get the next token in the module being analyzed.

The "main()" function shown in Figure 3 is the driver routine for the SCA. It starts the operation of the SCA by initially calling the parser, which in turn calls the scanner to get the "next" token. As parsing proceeds, the code implementing the token counting rules is executed. When the end of the Ada source code file is reached, main() calls the function "calc_metrics()" to compute the measures n_1, n_2, N_1, and N_2. Finally main() invokes "compute_timing_model()" to compute and print the compile time predictions.

```
 optional "c" code
 YACC variables and setup info
%%
 grammar productions
 prod1    { action1;  func1; }
 prod1    { action2;  func2; }
   .          .          .
 prodn    { actionn;  funcn; }
%%
  #include lex.yy.c
  #include y.tab.h
  main()
  {
    initialize table();
    yyparse();
    print measures and table();
    calc_metrics();
    compute_timing_model();
  }
  yyerror()
  {
  }
  calc_metrics()
  {
  }
  definitions of other functions
  compute_timing_model()
  {
  }
```

Figure 3. Contents of Ada.y

The output of YACC is the file named "y.tab.c" which contains the YACC-generated source code for the parser. The organization and contents of y.tab.c are summarized in Figure 4. The parser is contained within the "yyparse()" function. YACC constructs all the data structures and variables that the parser needs. When optional routines, like those which support the token counting strategy, are encountered in the text of Ada.y, YACC copies these routines verbatim (and any others found after the second set of percent signs) to y.tab.c for later compilation.

```
 "define" stmts from y.tab.h
 #include lex.yy.c
 main()
 {
    initialize table();
    yyparse();
    print_misc_operators();
    print_ident_table();
    calc_metrics();
    compute_timing_model();
 }
 yylex()
 {
 }
 yyerror()
 {
 }
 parse tables defines and declarations
/* parser for YACC output */
 yyparse()
 {
   yylex();
   yyerror();
 }
```

Figure 4. Contents of Y.tab.c

SCA Internal Operation. When the SCA is run, the input Ada source code module is read and analyzed token by token. The scanner, yylex(), operates as a subroutine to the parser. The parser invokes yylex() to pass it the code of the next valid token from the input file. If the scanner finds a lexically invalid token, it prints a message flagging the lexical error and continues scanning. But if the next token is valid, LEX passes the class or type of the token as a code to the parser. In addition to the token code, yylex() loads a parser-defined global variable, called "yylval", with the *attributes* of that particular token. The attributes of the token may be its value in the case, say, of a numeric literal, or the attributes may be a pointer to a location in memory in the case of a string.

Once the next valid token is received, the parser attempts to recognize or reduce the current syntactic construct of which the last several tokens passed are constituents. Whenever a valid construct is recognized, the token counting code associated with that construct is executed before parsing continues. When parsing continues, the parsers requests the next valid token. This cycle continues until all valid tokens have been processed. If at any time the parser cannot recognize (reduce) the current stream of tokens as a valid syntactic construct, the function call "yyerror(string)" is called, where the "string" parameter is an error message to the user. When the error function returns, the parser attempts to recover from the error and continue.

As tokens and other syntactic constructs are recognized, various individual counters as well as the Identifier Table are all continuously updated according to the rules of the token counting strategy. The Identifier Table eventually contains an entry (record) of every user-defined identifier found in the input source modules. Also stored in the Table are the type of each identifier and the number of times it occurred in that module as an operand or an operator or *both*.

When the end of the token stream is reached, parsing stops and the print_misc_operators() function prints a summary of the count of delimiters and single/multi-token operators. Next, the function "print_ident_table()" formats and prints each entry stored in the Identifier Table. Subsequently, the "calc_metrics()" function computes values for n_1, n_2, N_1, and N_2 by using the data from both the Identifier Table and each variable holding a count for an operator. Finally these values are input as independent variables into Equation 6 by the "compute_timing_model()" function.

The "compute_timing_model()" function prints a total of four environment-specific compile time predictions for the input module. Expressed in seconds, the values are considered to represent the time which would elapse if the input module were compiled in the given environment. The exact meaning of *time* in this context is explained later in this report.

In summary, the text of the two input files for the LEX and YACC programs together form the "source code" for the SCA.

Included in the YACC input are routines to implement the token counting strategy, build and update the Identifier Table, tabulate the Software Science measures, and eventually compute the compile time predictions. LEX produces a file which is "included" by YACC's input. YACC outputs a file of source code implementing the scanner, parser, and any user-defined routines. This source code is then compiled to produce the executable SCA.

IV. SCA Implementation Details

Chapter IV discusses in detail the routines responsible for data structure management and support of the token counting strategy. The measures tabulation and the timing model computations functions are also further detailed. The SCA driver, main(), and parser's error handling routine, yyerror(), were discussed in Chapter III and won't be discussed again here.

Data Structure Management and Token Counting Functions

The data used by the SCA are maintained in two types of data structures. The first is a globally-defined, integer variable. For each type of delimiter, single-token operator, and multi-token operator defined by the counting rules, an integer variable is declared to record the number of times that operator occurs in the source file. Since these variables are globally declared, any of the various functions implemented in the SCA can manipulate them. The declarations for the integer variables are grouped together and appear between the CFG and the main() function in the text of Ada.y.

The second type of data structure, the Identifier Table, records operand and operator counts. The Table is implemented as a singly linked-list of records, or *structures*, with each structure corresponding to a single entry in the Table. (A structure in the C language is a heterogeneous data construct similar to a record in other languages). The Identifier Table records each user-defined variable, literal, constant, subprogram, etc., all of which are collectively called "identifiers". Other information stored with each identifier includes the "type" of the identifier (i.e., is the identifier a variable, constant, literal, or subprogram?) and the number of times that particular identifier occurred as an operand and/or an operator in the input file. The C type declaration for an Identifier Table entry is shown in Figure 5.

```
struct ident_table_entry
{
char id[256];
char ident_type[11];
int ident_as_operand;
int ident_as_operator;
struct ident_table_entry *ptrnext;
};

 /* POINTERS TO MANAGE THE TABLE
ENTRIES */
struct ident_table_entry
*ptrfirst,*ptrthis,*ptrnew;
```

Figure 5. Identifier Table Entry

Table Management Functions. The SCA contains eleven functions to manage the Identifier Table in support of the token counting strategy. These functions are called during parsing to store, update, or search for data in the Table. A list of the Table management functions appears in Figure 6. The specific purpose and operation of these functions are described in the paragraphs which follow Figure 6.

```
initialize_ident_table()
install_ident(id,ident_type)
increment_operand_count(id,ident_type)
increment_operator_count(id,ident_type)
install_duplicate_name_ident(id,ident_type)
install_duplicate_name_ident2(id,ident_type)
is_ident_stored_as(id,ident_type)
decrement_operand_count(id,ident_type)
decrement_operator_count(id,ident_type)
install_loop_ident(id,ident_type)
print_misc_operators()
print_ident_table()
```

Figure 6. Table Management Functions

The initialize_ident_table() routine loads pre-defined Ada data types (e.g., Integer, String) as identifiers into the Table and records their type as a "data type". Since these pre-defined data types are usually not defined explicitly in the Ada source code, this pre-loading of the Table has the effect of establishing that certain identifiers (e.g., Integer, String) definitely occur in the role of data types. This bit of up-front information simply makes the job of typing any occurrences of these identifiers somewhat easier for the SCA.

Except for the table initialization and printing routine, the parser passes each function in Figure 6 a pointer to an identifier, id, and a pointer to the type of the identifier, ident_type. The identifier and its type are then stored as the next entry in the Table. For example, the function call

```
install_ident(my_identifier,my_ident_type)
```

would store my_identifier in the Identifier Table with my_ident_type as its type. To record the occurrence of my_identifier as an operand or an operator, either the increment_operand_count() or increment_operator_count() function is immediately called passing both my_identifier and my_ident_type as parameters. In this way, a given identifier (qualified by a specific type) as well as a record of its occurrence as an operand or operator are recorded in the Table.

If the identifier/type pair already exists in the Table, the install_ident() function wouldn't install it a second time. However, if one of the install_duplicate_name_ident() functions were used instead, the identifier/type pair would be unconditionally installed as the next entry in the Table. Install_duplicate_name() sets the operand count to one, and install_duplicate_name2() sets the operator count to one during the installation routine. Flexibility with identifier installation also permits an identifier or a data type which is *declared* more than once in the source file to be recorded as a distinct entry in the Table.

The context in which the identifier occurs in the text of the source code normally determines the particular "type" to assign to an identifier as well as whether to record its occurrence as an operand or an operator. However, there are cases in the Ada grammar where, for reasons having to do with the way the YACC program works, the SCA cannot determine the correct way to type and/or count an identifier when it is first encountered. For example, the SCA may be unable to determine whether to count an identifier as an operand or an operator at the particular point where that identifier is first recognized in the grammar. This problem can occur if the immediate context of the identifier in the source code gives no clue regarding the "type" of that identifier. In all such cases, the *Identifier Table* must be searched before making any determinations. Searching the table for the type of an identifier is the function of the "is_ident_stored_as()" routine. This function returns a value of 1 if the identifier/type pair passed as parameters are successfully found in the table. It returns a 0 value otherwise.

Sometimes the SCA is "forced" to guess at whether a particular token is used as an operand or an operator in a given context. Now suppose that further along in the "parse" of the syntactic construct containing that identifier it becomes evident that the original guess was incorrect, according to the rules. If the token is a delimiter, the integer variable recording its count can easily be decremented. But if the token is an identifier, more complex actions are required.

In the case of an identifier, if the SCA cannot correctly determine whether to increment the identifier's operand or operator count until more of the syntactic construct is reduced, then the SCA initially makes a "best guess" and increments one or the other of the counts on a tentative basis. Later in the parse when the SCA *can* determine the correct count to increment, the original guess might have to be corrected. To help correct the counts in the Table, the SCA can employ either decrement_operand_count() or decrement_operator_count() to reduce the operand or operator count of the original identifier by one. In this way, the integrity of the counting data in the Table is again preserved.

Concerning integer variables used in looping constructs, the counting rules stipulate that all such variables be counted as distinct operands. The install_loop_ident() function unconditionally installs a variable of this type as a "loop" variable and sets its operand count to one.

After parsing is complete, "print_misc_operators()" prints the count of each delimiter to standard output. A sample is shown in Figure 7.

```
There were 4 * operators in the module scanned
There were 3 + operators in the module scanned
There were 1 - operators in the module scanned
There were 22 / operators in the module scanned
There were 1 ** operators in the module scanned
```

Figure 7. Sample Output of Delimiter Counts

Finally, print_ident_table() formats the records of the entire Identifier Table and prints them to standard output. A portion of a typical Identifier Table is shown in Table 4.

Table 4. Sample Identifier Table

```
          THE SCA IDENTIFIER TABLE FOR THE INPUT FILE


IDENTIFIER           TYPE           TIMES USED AS
                                     OPERAND     OPERATOR


BOOLEAN              type              0            2
INTEGER              type              0            0
FLOAT                type              0            0
STRING               type              0            0
NATURAL              type              0            0
POSITIVE             type              0            0
DURATION             type              0            0
INSTRUMENT           var/const         2            0
NRPCA1               subprgram         2            0
PS_CS                var/const         2            0
PROCS                var/const         2            0
B                    var/const        15            0
TTRUE                var/const         1            0
T                    type              0            6
T                    type_conv         0            3
TRUE                 var/const         1            0
TFALSE               var/const         4            0
FALSE                var/const         1            0
TEST                 var/const         6            0
IDENT                unknownop         0            6
RECURSION            var/const         5            0
```

Measures Tabulation.

The function "calc_metrics()" totals the operator/operand counts for every token contained in the input source file. The tabulation of these measures depends on data drawn from both the Identifier Table and the integer variables which store the operator counts for delimiters. The resulting values of the Software Science measures n_1, n_2, N_1, and N_2 are recorded and printed to standard output. The calc_metrics() function determines the value of each of the measures according to the following four algorithms:

1. Set $n_1 = 0$.

   a. Sequence through each entry in the Table. If the operator count stored there /= 0, increment n_1 by one.

   b. For each possible delimiter, test the value of the variable storing the count for that delimiter. If the value /= 0, then increment n_1 by one.

2. Set $n_2 = 0$.

   a. Sequence through each entry in the Table. If the operand count stored there /= 0, increment n_2 by one.

3. Set $N_1 = 0$.

   a. Sequence through each entry in the Table. Add the operator count stored there to N_1.

   b. For each possible delimiter, add the value of the variable storing the count for that delimiter to N_1.

4. Set $N_2 = 0$.

   a. Sequence through each entry in the Table. Add the operand count stored there to N_2.

Timing Model Computation

As mentioned before, the function "compute_timing_model()" implements Miller's predicted compile time model which is repeated here for convenience:

$$T = K_i \ (N * \log_2(n))^{.4639} * ((2 * n_2) / (n_1 * N_2))^{.0745} \quad (6)$$

The measures n_1, n_2 and N_2, N, and n serve as independent variables in the equation. The length of the module, N, is given by $N = N_1 + N_2$. The vocabulary, n, is $n = n_1 + n_2$. The integer "2", originating in the definition of $L^{\hat{}}$, is an approximation for n_2^*, the theoretical minimum number of operators needed to execute a function.

The term $\log_2(n)$ causes the only difficulty. The library of math functions for C under UNIX version BSD 4.2 did not include a function to calculate the logarithm of an argument to the *base 2*. So the equality $\log_2(x) = \log_{10}(x) / \log_{10}(2)$ was used to convert the logarithm to the base 10 to the base 2. Since

$$c = (\log_{10}(2))^{-1} = 3.3219285 \quad (7)$$

$$\log_2(n) = c * \log_{10}(n) \quad (8)$$

The right side of Equation 8 is substituted for $\log_2(n)$ in Equation 6. Once again, since the value of T in Equation 6 is environment-specific, the value of T for each K_i listed in Table

3 is computed and printed to standard output. A sample of this output is shown below in Table 5.

Table 5. Sample Compile Time Predictions

```
PREDICTION FOR THE UNIX ASC ENVIRONMENT IS: 18.09 SECONDS
PREDICTION FOR THE AOS/VS ENVIRONMENT IS: 16.27 SECONDS
PREDICTION FOR THE VMS ISL ENVIRONMENT IS: 10.33 SECONDS
PREDICTION FOR THE VMS CSC ENVIRONMENT IS: 9.29 SECONDS
```

Samples of two source code files and their SCA output are provided in Appendix C.

V. Testing and Results

A discussion of testing the correctness of the SCA begins Chapter V. Considering the token counting rules as design specifications, caveats and deviations to the specifications are discussed next. The results of an analysis of the predictive ability of the compile time model is then presented and interpreted. Finally, a recalibration of the timing model is performed and the new results are analyzed.

Correctness Testing

The purpose of correctness testing is to verify that the SCA accurately counts every token in an Ada source module according to the token counting rules of Chapter II. The most important criterion of this testing is:

> For a given module, the SCA-produced values for the measures n_1, n_2, N_1, and N_2 should yield the same results as applying the token counting strategy manually. This means that the SCA must also type the tokens in accordance with the counting rules.

For the delimiters and single/multi-token operators, correctness testing occurred primarily during the development of the SCA. As a new capability of the SCA was added, it was tested and, if necessary, modified until it worked as intended. Thus correctness testing for operators was basically an iterative process of implementing code to count a specific type of operator and then applying the SCA to an input module which contained instances of that operator. The SCA-produced operator count was compared to a manual count of that operator in the input file. If the counts agreed, the next operator was selected for implementation in the SCA. If the counts disagreed, the cause of the inconsistency between the counts was discovered and corrected before continuing. In general, operators were selected in order from easiest to implement to most difficult.

The approach to correctness testing for *identifier* typing and operand/operator counts was more difficult but less structured than for delimiters and single/multi-token operators. This testing could only begin after the Identifier Table and its basic support routines worked. Testing now involved verifying that the SCA accurately recorded ***every*** identifier, its type, and its operand/operator counts.

Specifically, the tests involved verifying that the data in the printed Identifier Table was consistent with results from an exhaustive manual inspection of the input module conducted according to the rules of the token counting strategy. Every identifier was considered with respect to its context in the text of the source code. The typing and operand/operator counts for each identifier in the printed Table had to agree with those of the manual inspection. Since a given identifier can be an operand in one context and an operator in another, the entry for such an identifier had to reflect this as well.

Eighteen test files containing Ada source code were selected from the public domain for correctness testing. The files ranged in size from 2 kilobytes (about 100 lines) to 8 kilobytes. They included sophisticated constructs such as generics, tasking, entry/accept constructs, exceptions, and pragmas. The files were inspected and modified until as a group they contained one or more instances of each type of token and syntactic construct available in the Ada language (one exception is described in Caveats and Deviations, below). Every rule and special case in the token counting strategy was successfully tested and verified. Therefore, notwithstanding the exception discussed in the next section, the SCA accurately types and counts tokens according to the Miller and Maness counting strategy.

Caveats and Deviations

In some instances strict compliance to the token counting rules could be met marginally, or not at all. Fortunately, these are few.

Parentheses. Accurate parentheses typing depends on explicit intra-module declaration of identifiers which use parentheses. This means that if an identifier is used in an module but is not declared (defined) in that module, and if the identifier is used in conjunction with parentheses, then the specific *type* which is assigned to that set of parentheses may or may not be correct. For example, if an array object was imported from another module and encountered by the SCA as imported_array_object(i), this set of parentheses may be incorrectly typed as "invocation" rather than "dimensioning" parentheses. Regardless, the set of parentheses is counted as an operator in the usual way. In these cases a "best guess" at typing is employed. Since the total number of parentheses is always counted correctly, N_1 is correct. However, since the SCA may incorrectly assign parentheses types, it's possible that n_1 could be off by a small amount.

Global Variables and Types. Suppose two or more variables having identical names are declared in the same input module. If they are declared in separate statements, then each variable is counted as a distinct operand, as the counting rules stipulate. The same is true for types (e.g., my_type is String...). This feature does not apply to subprograms or packages, however. If a second subprogram or package has the same name as an earlier declared object of the same type, the second object is treated as a new occurrence of the original object. This rule extends to more than two occurrences of subprograms or packages as well. Regardless, the total occurrences for *any* object (identifier) is always accurate.

Families of Entries. The implementation in the SCA of grammar production 9.5 in Appendix E of the Ada LRM was changed to permit accurate counting of the identifier for the entry construct in the following manner. The original rule,

```
ENTRY_ identifier (discrete range) (formal part) ';'
```

was modified to the new form:

```
ENTRY_ identifier [discrete range] (formal part) ';'
```

Notice that the parentheses for the discrete range have been replaced by brackets. Although not shown here, both the "discrete range" and the "formal part" are optional in the production. The impact of this change for the SCA is important:

> Entry statements which use the discrete range portion of this construct must be modified to use brackets in place of the parentheses surrounding the discrete range.

Otherwise, the identifier of the entry will not be accounted for in the Identifier Table. Fortunately the discrete range option, which involves families of entry calls, is not commonly used in practice. Entry statements without the discrete range are correctly handled in the usual way. A possible work-around solution to this problem will be suggested in the recommendations of this report.

Generic Instantiations. Ada gives the user the ability to instantiate generic packages, procedures, and functions. The syntax of a generic instantiation includes an actual parameter list as an option. The rules explaining how the SCA categorizes these actual parameters as operands or operators are presented now.

For generic instantiations which pass actual parameters (e.g., ... IS NEW expanded_n(act_parameter_list)) the following rules apply:

1. If an identifier in the act_parameter_list is stored as a type or a subtype in the Identifier Table, the identifier is counted as an operator.

2. If an identifier in the act_parameter_list is stored as any other type the Table, it is counted as an operand. Obviously, this includes subprograms, variables, constants, literal, etc.

3. If an identifier in the act_parameter_list is <u>not</u> found in the Table, it is stored as a "var/const" type and counted as an operand.

<u>Overloading</u>. An identifier can have several alternative meanings at a given point in the program text; this property is called "overloading" (Booch, 1983). The effective meaning of an overloaded identifier is determined by the context (Booch, 1983). The SCA is not sensitive to overloading; it cannot discern the meaning or reference of a particular instance of an overloaded identifier based on the current context.

Nevertheless, every occurrence of an overloaded identifier will be recorded in *some* entry in the Identifier Table. If there are several existing Table entries for an overloaded identifier, the SCA cannot know *which* one of these entries to update when it recognizes an instance of that identifier. It simply updates the first such entry it finds during a sequential search of the Table. Consequently, any individual Table entry for an overloaded identifier may or may not be accurate in terms of its operand or operator count.

Fortunately, our strategy considers the data in the Identifier Table as an aggregate. When the values of the Software Science measures are tabulated, the correct operand/operator count for any identifier, overloaded or not, is obtained. Remember that the measures are defined in terms of totals and not in terms of individual identifier counts. Thus there is no adverse impact on the results.

SCA Testing Environment and Design

Correctness testing of the SCA is only concerned with how well the SCA implements the token counting strategy; it does not reveal any information regarding the accuracy of the tool as a predictor of compile time. But what about the predictive ability of the SCA? How well does the predicted compile time of the SCA match the actual compile time for a given Ada source code module compiled in a given hardware/software environment? These are the questions that the rest of this chapter will try to answer.

As discussed earlier, the SCA computes a prediction for compile time with respect to four different compilers in four unique computer environments. Only one of these four was selected to conduct compile time testing. We chose the ASC environment because it offered a familiar operating system (UNIX) and the Verdix Ada compiler. The UNIX "time" command was used with the "ada -v filename.a" command to allow the time expended during compilation to be recorded on a file-by-file basis.

At this point a brief digression concerning the type of software used during testing is appropriate. The title of this thesis states that the SCA is geared towards the prediction of compile time for *avionics* source code. The software we chose for this phase of testing came from a group of flight control and guidance software collectively called the Common Ada Missile Packages (CAMP) (User's Manual, 1987). The CAMP software consists of 250 files (about 16,000 lines of code). Slightly more than 200 CAMP files ranging in size from four kilobytes to 92 kilobytes were selected at random from the group. These files largely contained Ada generics designed for reuse in the development of missile guidance systems.

Because the CAMP constitutes a single system, there are compilation dependencies among the files; that is, many files "withed" other files. Therefore, we couldn't empty object code libraries between compilations. Unfortunately, a large number of components in the object code libraries increases overall compile time because the compiler has more information to search through during dependency checking. Even though Miller and Maness had used empty libraries in the development of their data base (Miller, 1986), we could not. The impact of this inconsistency on the validity of the results will be discussed in the analysis section of this chapter.

<u>Test Objectives</u>. The specific test objective was to produce both a value for predicted compile time and actual compile time for each of the CAMP files selected, and then to compare the

magnitude of the difference between the two values. Since all compiling took place on the ASC, all results are ASC-specific. An executable "shell" file invoked a sequence of steps in support of our objective. For each CAMP file in the test group, the shell file:

1. Submitted the CAMP file to the SCA as input and recorded the SCA's output in a designated output file.

2. Compiled the CAMP file and appended a record of the amount of time elapsed during compilation to the designated output file of step 1.

3. Searched the designated output file of steps 1 and 2 for the following information:

   a. name of the CAMP file being processed.
   b. name and version of the compiler used.
   c. date of the test run.
   d. the values of n_1, n_2, N_1, and N_2.
   e. the compile time prediction of the SCA.
   f. the actual compile time information recorded in step 2.

4. Copied the information found in step 3 to the next entry in a file called "Measures". Measures would eventually contain a similar entry for each CAMP file processed in the test.

5. (Optional) Deleted the designated output file of steps 1, 2, and 3, and deleted the current CAMP file being processed. (This step was necessary due to an installation-imposed restriction on disk space. Step 5 can be deleted if disk space is plentiful).

The text of the shell file implementing these steps is shown in Figure 8. All files involved in the test existed in the same directory on the ASC.

```
sca <$1.a >$1.o
time ada -v $1.a >>$1.o 2>&1
grep 'File: /en0/gcs88d/egoepper/camp' $1.o >>measures
grep 'Verdix' $1.o  >>measures
grep 'Sep ' $1.o >>measures
grep 'number of distinct operands: n2' $1.o >>measures
grep 'number of distinct operators:n1' $1.o >>measures
grep 'total number of operands:    N2' $1.o >>measures
grep 'total number of operators:   N1' $1.o >>measures
grep 'UNIX ASC ENVIRONMENT'  $1.o >>measures
grep 'real        ' $1.o  >>measures
rm $1.o
rm $1.a
```

Figure 8. Shell File for Testing Camp Software

A portion of the file Measures, which contains the results for the first two CAMP files processed, is shown in Figure 9. Notice that the "time" command outputs three types of times. According to the UNIX on-line manual, "real time" is comparable to "wall clock" time. "System time" can be viewed as the length of time the process responsible for compilation remains under control of the operating system. Finally, "user time" is the amount of actual CPU time consumed by the compilation process (and any sub-processes it spawned). These times are all measured in seconds to two decimal places.

The test ran on the ASC computer overnight when no other users were on the system in a "time share" mode. The test was also run during peak daytime hours and, interesting enough, the values for user and system time did not differ significantly.

```
File: /en0/gcs88d/egoepper/camp/S001000.a
Verdix Ada Compiler, Copyright 1984, 1985, 1986
compiled Thu Sep 15 11:48:44 1988
number of distinct operands:  n2 =    111
number of distinct operators: n1 =     36
total number of operands:     N2 =    126
total number of operators:    N1 =    541
PREDICTION FOR THE UNIX ASC ENVIRONMENT IS: 18.09
SECONDS
14.5 real         5.0 user         3.0 sys

File: /en0/gcs88d/egoepper/camp/S001001.a
Verdix Ada Compiler, Copyright 1984, 1985, 1986
compiled Thu Sep 15 11:49:20 1988
number of distinct operands:  n2 =     20
number of distinct operators: n1 =     21
total number of operands:     N2 =     21
total number of operators:    N1 =     78
PREDICTION FOR THE UNIX ASC ENVIRONMENT IS: 6.52
SECONDS
  10.5 real         1.8 user         2.5 sys
```

Figure 9. Sample Records in the Measures File

Data Aggregation. Selected data was extracted from the Measures file and used to build a spreadsheet using the commercially available software package "Quattro" by Borland, International. The data input into the spreadsheet included: CAMP filename and size in kilobytes; values of n_1, n_2, N_1, and N_2; SCA predicted compile time; and user/system time for actual compilation. The real time values were not included in the spreadsheet since they were not of interest in this work.

The spreadsheet, listed in Appendix A1, computes three additional data items for each CAMP file in the test. These three data items, listed in Table 6, were important to the analysis of the validity of the compile time model. The actual compile time is computed as the sum of the user and the system time, to keep consistent with the approach used by Miller and Maness. The prediction error is the difference between the predicted compile time and the actual compile time. CAMP file length, N, is the same as Halstead's definition from Table 2.

Table 6. Three Data Item Definitions

actual compile time = user time + system time
error = SCA predicted compile time - actual compile time
CAMP file (module) length = N = N1 + N2

Compile Time Results and Analysis. The spreadsheet lists both the predicted compile time values and the actual compile time values in the ASC environment for each CAMP file in the test. These data points are graphed in Figure 10 as lines. The predicted compile time is higher than the actual compile time by an average of roughly 4 seconds. Many different factors may be the cause of the higher values for predicted compile time. First, the ASC computer hardware, UNIX memory management, or UNIX job scheduling algorithms may have been modified during the two years since Miller published the environment-specific constant, K_i, for the ASC UNIX environment. Throughput improvements in any or all of these areas could decrease the values of the actual compile time data. A preliminary investigation indicates that there have not been "significant hardware changes" to the ASC since 1986 (Strovink, 1988).

Another possibility may be the difference between the type of software used for the current tests and the type of software used in the derivation of the timing model. The modules which Miller used came exclusively from the Ada Compiler Validation Capability (ACVC) test suite. This is a group of compiler test modules written to specifically evaluate how well a vendor's compiler adheres to the specifications of the Ada LRM. The ACVC test suite includes some very large files; they are much larger than any of the CAMP files used in the test (Shaw, 1988).

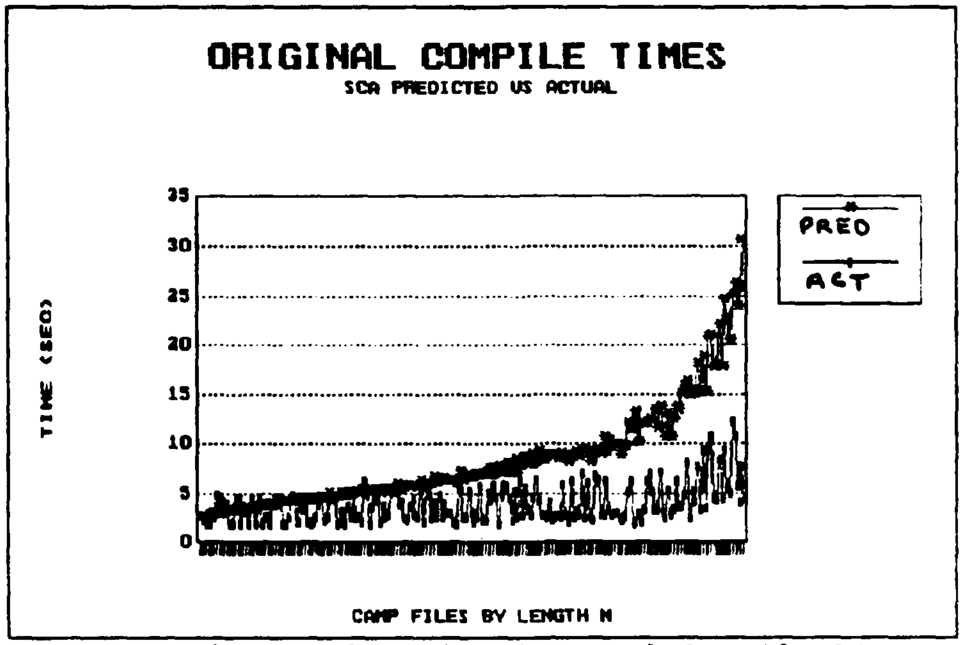

Figure 10. Predicted and Actual Compile Time

Further, the CAMP modules, which produced all of the predicted compile time values in this effort, are "real world", production-quality, missile guidance and control software. In fact, most of the CAMP modules are Ada generics, including generics nested within generics in many cases. Needless-to-say, there is probably quite a number of differences in these two groups of software.

A third possibility involves "compilation dependencies"; that is, the problem of certain compilation units depending on the previous compilation of other units. Miller could "clean out" his compilation libraries before each compile time data point was recorded. For the reasons mentioned earlier, our tests could *not* do this. Our non-empty libraries may have *decreased* the value for actual compile time in some cases, since certain parts of the source code in the current file may have been compiled previously with other CAMP files. This important aspect of our testing procedures and its impact on the results will be discussed again in a later section.

Finally, a fourth possibility is the substitution of $\hat{L}$ for V^*. Recall that this was done to adapt the compile time model to the measures which the SCA produced. Perhaps it constitutes a misapplication of the model. That seemingly insignificant change could indeed throw the predicted versus actual compile time results off by a constant scaling factor (Shaw, 1988). This issue too will be revisited.

Whatever the cause of the higher predicted compile times, we must consider the following possibilities: the model is actually correct but there have been environmental changes to the ASC which explain the average prediction error; *and/or* the file sizes of the CAMP software used in the test were not a representative sample; *and/or* there were shortcomings in the testing procedures which introduced error; *and/or* we misapplied the compile time model when $L^{\hat{}}$ replaced V^{*}.

Prediction Error. The spreadsheet computes the difference between the predicted compile time and the actual compile time for each CAMP file in the test. We'll call this difference the "compile time prediction error" of the SCA in the ASC environment for a particular CAMP file, or just "error" when the context is clear. Statistics for the error values are shown in Table 7.

Table 7. Prediction Error Statistics

mean = μ = 4.35
variance = σ**2 = 19.51
standard deviation = σ = 4.42

We know from elementary statistics that 95% of the prediction error values must lie within the interval $(\mu - 2\sigma, \mu + 2\sigma)$. That is, we can be 95% certain that the error of the prediction lies within the interval (4.35 ± 8.84). This range of values can be viewed as a *confidence interval* for the SCA compile time prediction. In other words, we can be 95% confident that any error in the SCA's timing prediction must lie within the range (4.35 ± 8.84).

There is a "quick fix" for Equation 6 so that compile time predictions for the ASC environment have an average error close to zero. Since the average error is positive (4.35), if we subtract it from the original value of the SCA prediction, then we could be 95% sure that the error of the modified prediction is, on the average, very close to zero. Thus the value of the average error can be used to adjust the SCA compile time model by subtracting $\mu = 4.35$ from the value of T in Equation 6 to produce a modified SCA compile time prediction, $T^{\hat{}}$, as shown below in Equation 9:

$$T^{\hat{}} = [K_1(N*\log_2(n))^{.4839} * ((2*n_2)/(n_1*N_2))^{.0745}] - 4.35 \qquad (9)$$

<u>Impact of the Data Distribution on Error</u>. There were over 200 CAMP files used in the compile time model tests. Characteristic of most of these files was that the actual and predicted compile times were relatively low (0 - 7 seconds). This resulted in the following phenomenon: the majority of data points representing predicted/actual compile time information are clustered. Observe in Figure 11 that the data points tend to fall in the lower left-hand corner of the graph, with very few of them in the upper right-hand corner. The data point clustering reflects that most of the test files were relatively small and had relatively low values for predicted and actual compile time.

Because of the clustering of the data points, the current values of the parameters a and b in Miller's compile time model must be re-adjusted with respect to our data. Miller's original values of a and b were derived from a data base built using a better balance of large and small files (the ACVC test suite). That is, Miller's data represents a more even distribution between files having short compile times and those having relatively longer compile times. Our data does not have a balanced distribution of compile times in this sense. The difference in the characteristics of the data used in Miller's work and this effort forms the basis of the strongest argument for a recalibration of the parameters of the model.

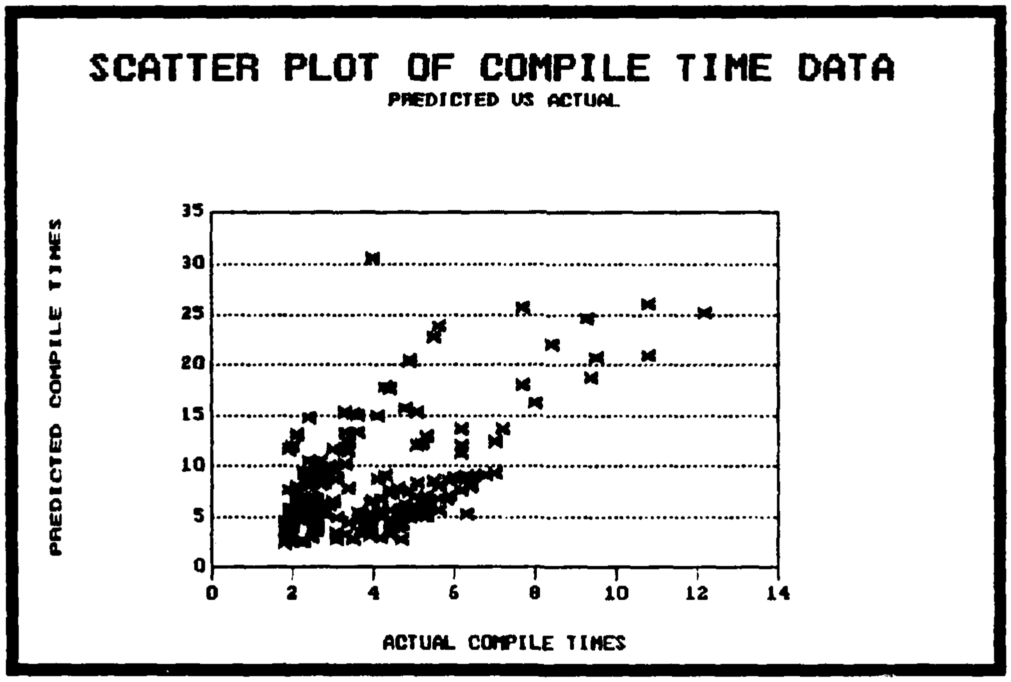

Figure 11. Data Point Distribution

The Impact of File Length on Error. Returning to the last column of the spreadsheet in Appendix A1 again, we want now to know how the length, N, of the file affects the magnitude of the prediction error values. Recall that the length of a file is the sum of the total number of tokens in that file (less comments); that is, $N = N_1 + N_2$. We can observe the impact of file length on prediction error in Figure 12. The CAMP files, whose lengths form the x-axis, were sorted in ascending order by size of N before plotting the corresponding error values. Notice how the magnitude of the prediction error increases as the value of N increases. Figure 12 clearly shows that as N increases, the compile time prediction becomes increasingly inaccurate. Why does this happen?

One reason the compile time prediction becomes less accurate as N increases may have to do with a particular shortcoming of the test procedures already mentioned. Failure to clean out the compilation dependency and object code libraries could, in fact, distort the results because Miller based his actual compile time data on "fresh" libraries (Shaw, 1988).

To understand how this works, suppose a test procedure allows the library to fill up with the object code of previously compiled units. Now, suppose a relatively large file "withs" one or more of these compiled units from the library. During *actual* compilation of the large file, any objects encountered which are specifically made visible by dependency on an earlier compiled unit need *not* be recompiled (because the object(s) is already compiled into object code in the library unit). The SCA, however, cannot make an allowance for previously compiled objects made visible by a "withed" unit. The SCA bases its prediction on the assumption that every construct in the text of the file (except comments) has to be compiled in the current compilation. Obviously, in this case the predicted compile time should be much higher than the actual, even for a perfect compile time model. And higher predictions, especially among the larger files, are exactly what we found.

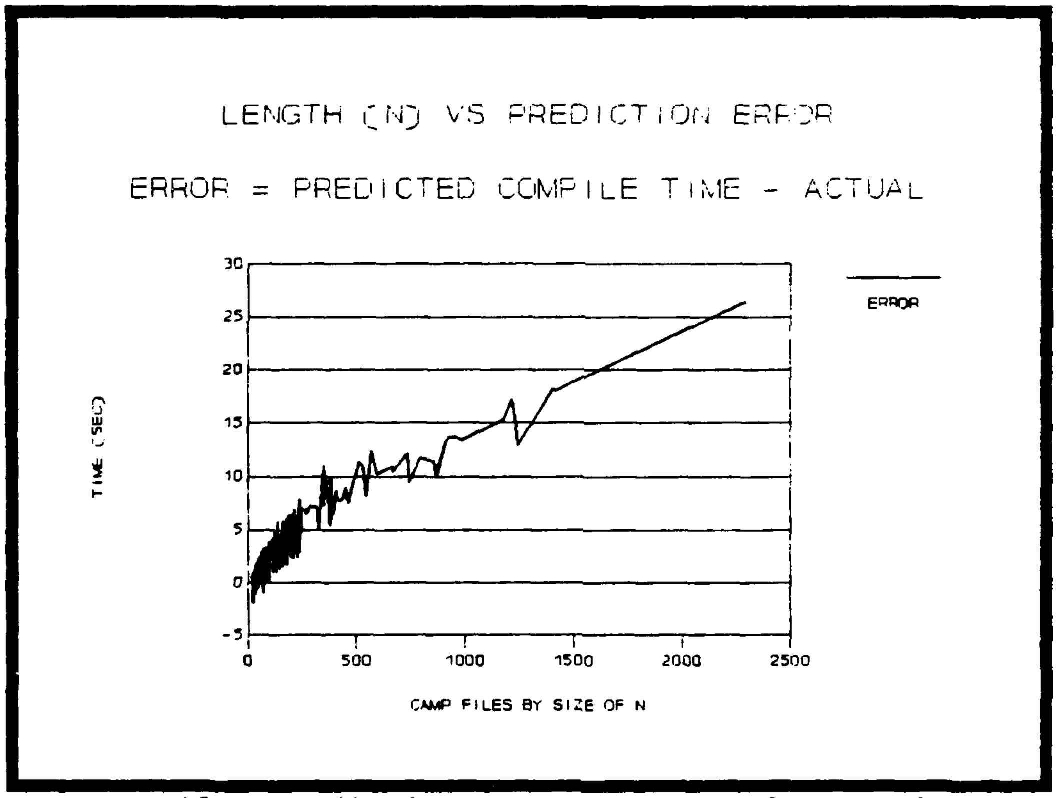

Figure 12. Prediction Error Versus File Length (N)

Perhaps Miller's model relies heavily on the magnitude of N for its predictive ability precisely because the object code libraries were cleaned out between each compilation. That would explain why the model is so sensitive to the value of N. Sensitivity in this sense means that small changes in the value of N may cause large changes in the value of T. In fact, by inspection of Equation 6 we can see that the term involving N, shown below, is the dominant term in the model.

$$(N * \log_2(n))^{.4839} \tag{10}$$

Another possibility that may explain the higher predicted compile time values involves the Ada compiler itself. If a newer, more efficient version of the Verdix compiler has replaced the version which was used by Miller, actual compile time values would change. If this new version is faster than the earlier version, then the effect of file length on actual compile time would be considerably less on the ASC today than it was two years ago. A preliminary investigation revealed that there were upgrades to this compiler since 1986, thus possibly explaining the behavior in Figure 12 (Strovink, 1988).

If either of the last two explanations for the higher predicted compile time values has merit, then recalibrating the compile time model should cause a change in the magnitude of the exponent a = .4839. The exponent should decrease as a result of recalibration so that the impact of N on the compile time prediction T is reduced.

The Impact of File Length on Actual Compile Time. Figure 13 below reveals further evidence that the model should be recalibrated. The graph indicates that actual compile time may not be a strictly increasing function of module length N. Indeed, the function behaves as if N was a factor in predicting actual compile time, but not as dominating a factor as Equation 6 expresses.

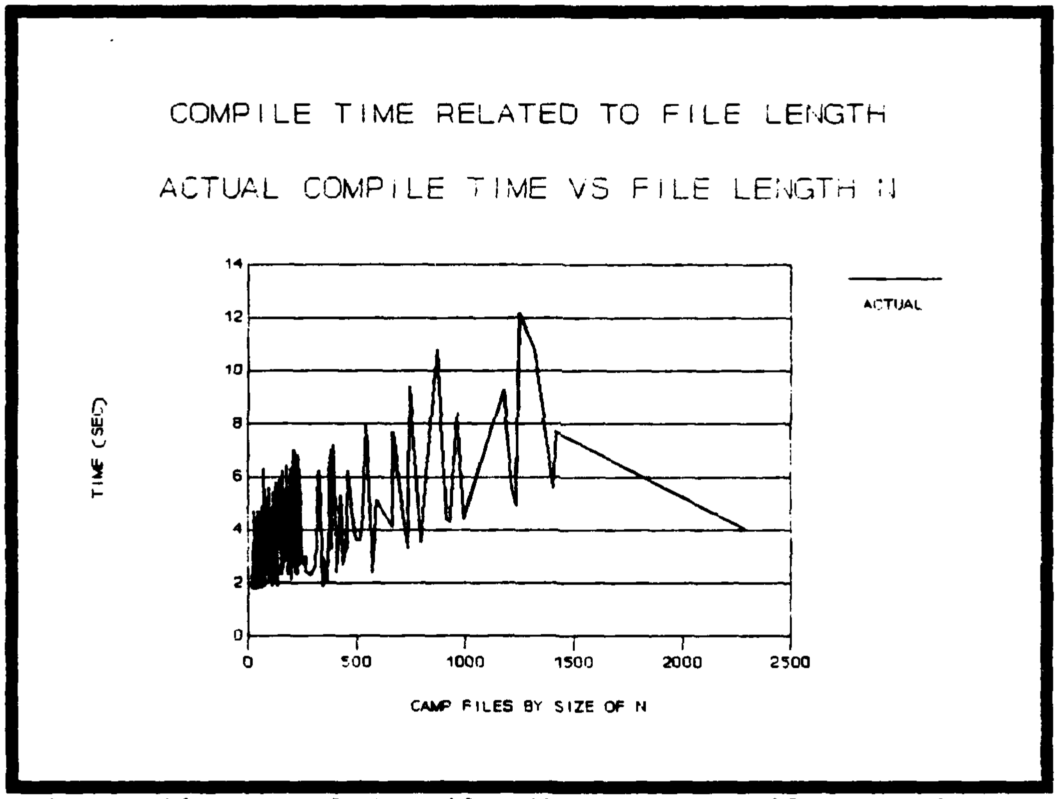

Figure 13. Actual Compile Time Versus File Length (N)

Let's consider the data points greater than 1500 tokens. The actual compile time for these modules *decreases*. Why? If the data are valid (i.e., the experimenter has not made an error), then can we conclude that, in some cases, actual compile time is more heavily influenced by some factor other than file length? This is a possibility. But it is also very likely that error was introduced into the measurement of actual compile time during the testing process itself. There are several ways this could have occurred. The UNIX command "time" is known to suffer from certain inaccuracies and round-off error, and recall that all the actual compile time values were based on the output of this command.

Model Recalibration

As the analysis in this chapter suggests, the compile time model should be recalibrated, meaning that the exponents or parameters of the terms in Equation 4 should be re-estimated for the current data and environment of the ASC. After performing linear regression on the parameters a and b of Equation 4 using Quattro, the exponents became a = .2683 and b = .2609. Hence, the timing model can now be expressed as:

$$T' = e^{.3522} * (N*\log_2(n))^{.2683} * ((2*n_2)/(n_1*N_2))^{.2609} \qquad (11)$$

The details of the recalibration can be found in the spreadsheet in Appendix A2. Notice in Equation 11 that the new parameters are significantly different from the original ones. As expected, the magnitude of the exponent for the term involving N has decreased. Interestingly, the value of the other exponent has increased almost fourfold.

Naturally, Equation 11 was examined to see if it would predict compile time better than Equation 6. The results of this examination are listed in the spreadsheet found in Appendix A3. Values for the compile time predictions using the recalibrated model are plotted against the actual compile times in Figure 14. Clearly, the recalibrated model fits the actual compile time data much better than the original model.

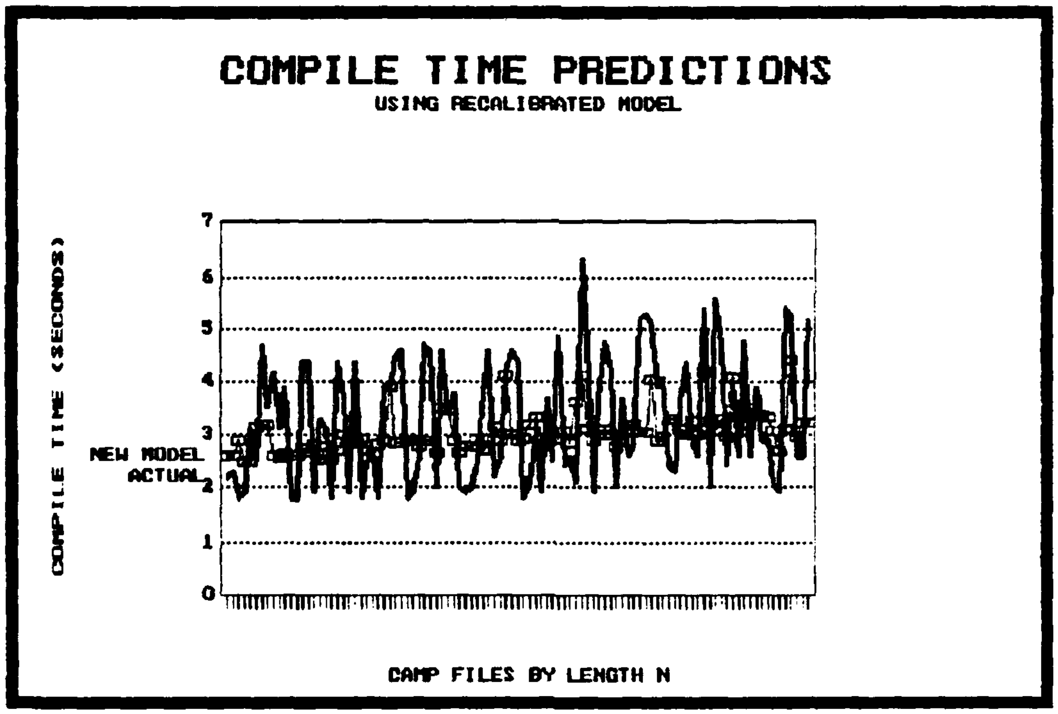

Figure 14. Comparison of Predictive Models

A summary of the results of the linear regression, located on the last page of the Appendix A2, includes other information about the recalibrated model. Specifically, the goodness of "fit" index, R-squared, which ranges between zero (poor) and one (good), is given as .3083. The reason for this relatively low value is related to the preponderance of data points clustered together towards one end of chart in Figure 11. Precisely because the data is "lop-sided" in this way, it is difficult or impossible to statistically find an expression for a mathematical function which reflects the behavior of these data points without a large margin of error. Thus, for the particular CAMP files selected in the test, the behavior of the compile time data precludes a perfect fit for any model we could derive using this data alone.

This is not to say that the recalibrated model lacks validity. On the contrary, it fits *our* data well, as the statistics in Figure 15 reveal. Since the average error for the original model is $\mu = 4.35$, but the mean error for the recalibrated model is only $\mu = -.25$, it has a great deal of credibility.

MEAN ERROR	=	-.25
VARIANCE	=	2.03
STND DEV	=	1.43

Figure 15. Recalibrated Model Statistics

After substituting Equation 11 for Equation 6 in the code for the "compute_timing_model()" function, we can assert with 95% confidence that the average difference between the SCA's recalibrated prediction, T', and the actual compile time is near zero. Moreover, the 95% confidence interval now becomes:

$$(\mu - 2\sigma,\ \mu + 2\sigma) \tag{12}$$

$$= (-.25 - (2*1.43),\ -.25 + (2*1.43)) \tag{13}$$

$$= (\ -3.11,\ 2.61) \tag{14}$$

Therefore, the SCA will predict compile time such that 95% of the error values will fall in the interval (-3.11, 2.61). This is what was meant earlier in this report by the phrase "*range* of predicted compilation times".

In short, we have observed that recalibration of the parameters significantly improved the predictive ability of the compile time model. The small mean error value of the recalibrated model gives credence to the notion that the model rests on a solid theoretical foundation. However, before the low R-squared value leads us to doubt these assertions, a series of more realistic and comprehensive tests using more "balanced" data should be performed. Only then can we begin to draw definitive conclusions regarding the empirical validity of the basic compile time model (Equation 4).

VI. Conclusions and Recommendations

Verification of the SCA

The merits of SCA should be judged according to how well it implements the token counting strategy of Chapter II and the criterion on page 45, because the primary goal in building the SCA was to automate this strategy as completely and accurately as possible. More precisely, the SCA's main purpose is to accurately count operators and operands according to the token counting rules. The only area where the SCA deviated from the rules was in the grammar for "families of entries" involving parentheses. In fact, the SCA counts the tokens correctly for entry families; however, the source code must be slightly altered prior to scanning.

The impact of other exceptions noted in the section "Caveats and Deviations" is minor. That section explains where the SCA does not, or *cannot*, adhere precisely to the rules. So certain token counting problems as, for example, identifying the "type" of a variable which is declared in a "withed" compilation unit, cannot currently be handled by the SCA. Nonetheless, the SCA succeeds in correctly computing the Software Science measures n_1, n_2, N_1, and N_2 according to the token counting strategy of Chapter II. This is the most important outcome of this work.

Validity of the Recalibrated Timing Model

The recalibrated compile time model was tested in the context of a unique computer environment (ASC) for a particular type of software (CAMP) contained in files whose compile time distribution had specific characteristics (clustered near low values). In *this* scenario, the recalibrated model was valid; we can say with 95% confidence that the average error is close to zero. Furthermore, we can assert with the same confidence that all prediction errors should fall inside the interval (-3.11, 2.61).

Applicability of SCA

The SCA is widely applicable; the recalibrated compile time model implemented in the SCA probably is not. Both the model and the SCA *would be* applicable for predicting compile times for a group of Ada software modules on the ASC having the following two characteristics:

1. The modules have relatively short compile times.
2. There are compilation dependencies among the modules.

Unfortunately, the compile time model, as it is currently calibrated, probably has limited universal applicability. By "universal applicability" we mean the model would most likely have to be recalibrated each time the SCA was ported to a new environment. Porting instructions are listed in the User's

Manual in Appendix B, and recalibration can be accomplished exactly as it was done in this report.

There are at least two areas where the SCA could be used in software research. First, it could be used by software metrics researchers as an automated token counting tool to compute Software Science measures. Since it allows researchers to circumvent painstaking manual counting approaches, a great deal of metrics data could be accumulated quickly and easily using the SCA.

The other area where the SCA could prove useful is the evaluation of Ada compilers. Let's assume that the compilation timing model has been recalibrated for a specific environment; it predicts compile time with an error near zero over a reasonably large confidence interval. Let's also assume that the model is tailored to avionics software; that is, the model was derived from and validated against a compile time data base built using avionics software. Then researchers interested in evaluating the compile time efficiency of a newly developed Ada compiler on the market could use the SCA in the following way.

A representative test suite of avionics software could be submitted both to the SCA and the new Ada compiler. If the actual compile times for the test suite varied markedly from the predicted compile times, then the magnitude of the differences would be a reflection of the relative compile time efficiency of the new compiler with respect to the compilers used in the calibration of the model. In other words, researchers could tell if a new compiler was faster or slower *on average* than the compilers already in use.

Conclusions

This section discusses some specific conclusions and conjectures which can be made as a result of this work.

1. The programming task is made much more efficient using program generation tools like LEX and YACC. The success of this effort is a direct consequence of using these tools to help develop the SCA. Programmer productivity can be increased significantly through program generation tools of this type.

2. Ada is a language rich in semantic characteristics. A token counting strategy (like the one implemented in this work) which depends on semantic context is necessarily a challenge to automate. The nuances of the counting rules in a context-based strategy are difficult to capture in an automated analysis tool. As a result, human decision-making ability could not always be emulated in the SCA.

3. Models to predict the compile time of Ada modules are not independent of the computer environment in which they are used. Indeed, the predictive ability of a compile time model is sensitive to changes in the efficiency of the environment in which it is based. Any compile time model should be calibrated for the specific machine and operating system environment, type of software to be compiled, and particular version of the compiler to be used. Furthermore, the modules used in the calibration should be drawn from a set whose sizes form a uniform distribution over a large range. Finally, the set of modules used in the calibration should either include compilation dependencies or not allow them at all.

4. With the last paragraph in mind, we can probably say that there is not a single compile time model for Ada modules which can be universally applied in all applications.

Recommendations

This section begins with a discussion of ways to expand or improve the SCA software tool itself. These follow-on enhancements are concerned with the quality of the SCA *per se* and not so much with how the SCA is used. Next, we'll outline some activities which might use the SCA to explore areas of interest to developers of avionics software using Ada. Finally, a project to improve the accuracy and applicability of the SCA's compile time predictions is outlined. This last effort is fairly ambitious, but it may be the highly profitable to pursue for researchers of compilation timing models and Ada software developers.

SCA Enhancements. The section entitled "Caveats and Deviations" discussed several areas where the SCA falls short of a complete and unequivocal implementation of the token counting strategy. The most important of these areas is the problem with "Families of Entries", because Ada source code which contains entry families must be modified prior to processing by the SCA.

A solution for this problem should be found. One possibility involves reworking the applicable production in the grammar to allow the use of parentheses for entry families (as it originally did). We could not find a way to do this and still implement the token counting strategy properly. Another possibility might be to pre-process the Ada source file checking for the use of entry families. The parentheses in any use of entry families would be substituted with brackets. The pre-processor and SCA could be run consecutively from a command file.

The other topics listed in "Caveats and Deviations" are less of a problem. In fact some of them, like the parentheses problem, are probably benign. In any case, the other areas in that section should be re-examined. Some token counting rules may not yield to automated methods, however. For example, it may be impossible for the SCA to handle object overloading as precisely as the strategy requires. Nevertheless, if strict adherence to the letter and spirit of the token counting rules is the prime objective, the specific areas to begin scrutinizing the SCA are identified in that section.

Another idea would be to compute the value of n_2^* in the SCA. This would permit the use of the canonical version of the compile time model. Also, computing all the metrics of Table 2, "Halstead's Equations", would be beneficial to researchers. Like the compilation timing model implemented in the SCA, these metrics are based on the Software Science measures n_1, n_2, N_1, and N_2. (The current version of the SCA has been enhanced to produce the following Software Science metrics: total token count N, volume V, estimated length $\hat{L}$, and estimated effort $\hat{E}$).

Implementing other equations for compile time modeling in the SCA could be productive. If the model is based on n_1, n_2, N_1, and N_2, then it could easily be implemented in the SCA. In fact, it may be possible to integrate a new compile time model which was not computed using n_1, n_2, N_1, and N_2. Such a model may have a different theoretical basis altogether. Specific procedures for adding new metrics and models are covered at the end of the User's Manual in Appendix B.

Employing the SCA. One of the disappointing aspects to the validity testing of the timing model implemented in the SCA was the lack of authentic avionics software with which to test. It would be interesting to see if the results would change significantly if our testing was duplicated using authentic avionics software in place of the CAMP software. If all other factors remained constant, significant changes would probably occur in the test results only if:

1. There are radical differences in the composition of avionics software and missile guidance software used in the test.

2. The compile time model is sensitive enough to these differences to alter the value of T.

We conjecture that both conditions 1 and 2 are false. But it would be simple enough to duplicate our testing procedures using avionics software to answer this question with certainty.

One of the other issues left unaddressed in this effort involves the identification of the specific Ada constructs most responsible for compilation overhead. In other words, we still do not know to what extent specific programming constructs (e.g., tasks, generics, pragmas) of the Ada language affect compile time. The SCA can be used to help answer this question. The code in the parser can easily be modified to record the occurrences of specific language constructs. The counts of these constructs could be statistically correlated with compile time data for the module(s) containing the constructs. In this way, the impact of these specific Ada constructs on compile time could be explored and possibly explained using the SCA as an investigative tool.

Follow-on Project. The last recommendation is an ambitious follow-on project extending the applicability of the compile time predictions of the SCA. The goal would be to recalibrate the compilation timing model, but this time using a much larger and more varied data base of Ada source code modules.

The first step would be to build a data base of Software Science measures and actual compile times. The measures n_1, n_2, N_1, and N_2 could easily be obtained by scanning a large number of modules of widely varying lengths with the SCA. Each module would then be compiled by several different validated Ada compilers in the context of a single hardware/software environment. The actual CPU time elapsed during the compilation process would be automatically recorded for each file. Best results would be obtained if test modules were drawn from pools of several different types of "real-world", production-quality Ada source code (e.g., avionics, data processing, data base software, etc).

The data base would be used to derive values for parameters a and b of the basic theoretical model for compile time (Equation 2). The validity of the new compile time model should be tested in the same way and for the same reasons Miller's original model was in the course of this study. Predicted and actual compile times must be compared, and any untoward tendencies of the model's behavior must be examined. If the timing model is well-behaved and the R-squared value is reasonably close to 1, we think the model should be retro-fitted into the original SCA. The final product would be a useful tool for metrics researchers and software developers alike.

Appendix A1: Analysis of Original Compile Time Model

CAMP FILENAME	SIZE (K)	n2	n1	N2	N1	LENGTH N
S681240.A	4	3	12	4	14	18
S681230.A	4	3	12	4	14	18
S644001.A	6	3	7	4	15	19
S002B00.A	6	3	15	5	17	22
S001700.A	6	3	15	5	17	22
S682W00.A	12	6	8	7	16	23
S662001.A	4	4	7	5	19	24
S651001.A	4	4	7	5	19	24
S671500.A	6	4	16	6	18	24
S001800.A	8	4	15	7	19	26
S361001.A	8	4	15	7	21	28
S002D00.A	8	4	16	7	21	28
S002E00.A	8	4	16	7	21	28
S671300.A	6	6	18	9	22	31
S671400.A	6	5	15	9	22	31
S002800.A	8	6	16	9	23	32
S001200.A	8	4	16	9	23	32
S001500.A	8	6	17	9	23	32
S002400.A	8	4	16	9	23	32
S671200.A	6	5	16	9	24	33
S361000.A	8	5	16	6	27	33
S644121.A	8	6	16	9	25	34
S671100.A	8	7	17	11	24	35
S002J00.A	6	7	17	10	26	36
S687E00.A	6	6	18	12	27	39
S687D00.A	6	5	16	12	28	40
S002K00.A	8	7	18	12	28	40
S001400.A	10	8	18	13	27	40
S615000.A	6	12	9	12	30	42
S661600.A	10	8	20	13	29	42
S661A00.A	10	8	21	13	31	44
S002F00.A	8	7	16	14	32	46
S002G00.A	8	7	16	14	32	46
S687F00.A	8	8	22	15	31	46
S661810.A	8	8	20	14	34	48
S661820.A	8	8	20	14	34	48
S002C00.A	10	6	20	14	34	48
S661800.A	6	10	15	11	37	48
S682Z00.A	30	11	11	17	33	50

CAMP FILENAME	SIZE (K)	n2	n1	N2	N1	LENGTH N
S001300.A	12	9	22	16	34	50
S002500.A	10	6	20	15	35	50
S002100.A	8	7	20	15	36	51
S002200.A	10	7	20	15	36	51
S682S00.A	8	10	19	19	32	51
S652200.A	8	7	24	16	39	55
S684200.A	8	9	21	17	38	55
S682Q00.A	8	13	19	22	34	56
S687A00.A	8	7	19	15	42	57
S613000.A	6	17	11	17	40	57
S661700.A	8	8	17	17	40	57
S653200.A	8	9	24	17	42	59
S002I00.A	8	9	21	19	41	60
S644240.A	12	15	24	21	39	60
S634000.A	10	12	17	19	43	62
S002H00.A	8	8	18	19	43	62
S652300.A	8	8	24	19	44	63
S682V00.A	10	14	23	25	40	65
S662100.A	20	8	25	16	50	66
S682P00.A	8	14	23	26	41	67
S687C00.A	8	11	31	21	46	67
S644120.A	16	14	21	15	52	67
S686001.A	8	12	12	13	58	71
S684300.A	10	12	23	23	49	72
S002300.A	10	9	22	23	50	73
S651300.A	20	8	23	17	56	73
S684500.A	10	12	24	23	51	74
S653300.A	10	12	29	23	52	75
S002A00.A	10	9	20	30	45	75
S652100.A	8	10	26	24	51	75
S687300.A	14	10	19	21	55	76
S687400.A	14	10	19	21	55	76
S686200.A	14	9	21	21	56	77
S686300.A	14	9	21	21	56	77
S671001.A	6	16	16	17	63	80
S652600.A	10	10	27	24	56	80
S653600.A	10	11	29	24	58	82
S682N00.A	8	18	23	33	50	83
S682R00.A	8	18	23	33	50	83
S651200.A	22	9	23	20	65	85
S653100.A	10	13	28	29	57	86
S687800.A	16	10	19	23	66	89
S687G00.A	8	13	26	32	58	90
S684001.A	6	18	17	19	73	92
S002900.A	12	15	23	32	62	94
S686400.A	16	10	24	25	69	94
S684100.A	10	15	21	34	60	94

CAMP FILENAME	SIZE (K)	n2	n1	N2	N1	LENGTH N
S682T00.A	8	13	25	38	59	97
S001001.A	8	20	21	21	78	99
S682L00.A	16	15	22	33	66	99
S661400.A	12	16	31	32	70	102
S661320.A	12	19	24	34	71	105
S001100.A	22	15	23	32	75	107
S687500.A	16	13	21	31	78	109
S687600.A	18	11	20	28	82	110
S002600.A	12	14	26	39	71	110
S644170.A	10	9	20	49	61	110
S686100.A	18	13	27	35	81	116
S661001.A	8	23	19	24	93	117
S681500.A	6	12	22	47	70	117
S682E00.A	14	19	31	44	74	118
S671600.A	16	16	27	40	79	119
S687900.A	20	16	23	34	85	119
S622001.A	8	20	23	42	79	121
S682M00.A	18	20	23	45	76	121
S661520.A	14	18	32	44	80	124
S644230.A	16	21	26	47	79	126
S002700.A	14	18	21	49	77	126
S653001.A	6	25	19	26	100	126
S001600.A	32	16	26	36	91	127
S686500.A	20	15	25	36	92	128
S652001.A	6	25	21	26	103	129
S644100.A	8	27	23	28	105	133
S661530.A	14	14	28	46	93	139
S686800.A	24	17	23	43	100	143
S687001.A	6	29	19	30	116	146
S651100.A	32	16	32	39	108	147
S644150.A	16	14	20	67	83	150
S661500.A	14	31	21	32	125	157
S681600.A	6	12	23	68	91	159
S686700.A	24	19	24	49	111	160
S632000.A	12	39	26	46	115	161
S672000.A	14	34	19	36	126	162
S687700.A	26	16	21	47	127	174
S631000.A	14	42	26	49	125	174
S634001.A	36	16	20	53	122	175
S686600.A	24	21	24	55	122	177
S602000.A	14	40	23	42	138	180
S682C00.A	18	26	35	71	109	180
S682U00.A	12	27	38	72	110	182
S682X00.A	12	35	30	61	122	183
S661300.A	14	35	27	52	131	183
S644220.A	20	31	31	63	120	183
S611000.A	10	34	21	62	126	188

CAMP FILENAME	SIZE (K)	n2	n1	N2	N1	LENGTH N
S682B00.A	18	27	36	76	114	190
S612000.A	10	34	21	62	129	191
S652500.A	22	29	37	70	122	192
S661310.A	14	27	27	65	130	195
S644210.A	12	18	25	79	117	196
S653500.A	24	30	37	72	124	196
S682Y00.A	14	35	31	66	136	202
S682G00.A	30	25	25	70	133	203
S682900.A	18	24	31	74	129	203
S684400.A	18	40	39	70	139	209
S644200.A	24	36	37	72	142	214
S652400.A	20	26	31	82	135	217
S644160.A	16	13	21	101	117	218
S644110.A	18	17	29	92	134	226
S661510.A	20	28	38	80	147	227
S653400.A	22	27	31	88	141	229
S682700.A	22	24	33	91	138	229
S682001.A	18	48	23	49	188	237
S686900.A	26	24	27	80	157	237
S614000.A	12	45	15	81	158	239
S682F00.A	18	29	36	106	148	254
S682A00.A	20	32	38	110	154	264
S682D00.A	20	32	38	110	154	264
S644130.A	16	14	25	118	147	265
S644180.A	16	19	23	123	163	286
S644122.A	22	18	27	134	175	309
S651000.A	30	51	24	65	253	318
S686A00.A	44	44	48	108	218	326
S632001.A	6	48	38	116	229	345
S002001.A	14	74	25	75	273	348
S681400.A	12	17	26	151	198	349
S631001.A	6	50	38	120	241	361
S682H00.A	38	45	38	133	236	369
S686B00.A	34	48	50	127	245	372
S662300.A	52	58	46	119	260	379
S661900.A	36	48	35	131	250	381
S683000.A	22	65	30	84	297	381
S682100.A	38	33	36	141	243	384
S671000.A	42	63	23	71	321	392
S682800.A	24	32	36	173	232	405
S644140.A	14	14	26	182	226	408
S602001.A	14	52	38	132	292	424
S681700.A	8	12	23	206	229	435
S687200.A	24	34	37	177	276	453
S672001.A	56	61	37	146	313	459
S687100.A	28	47	43	181	310	491
S623000.A	22	80	48	136	375	511

CAMP FILENAME	SIZE (K)	n2	n1	N2	N1	LENGTH N
S688000.A	80	85	36	106	423	529
S684000.A	46	89	31	104	440	544
S623001.A	12	52	52	146	423	569
S682600.A	80	71	47	195	397	592
S682J00.A	44	45	40	258	406	664
S001000.A	64	111	36	126	541	667
S687H00.A	16	41	44	287	447	734
S681000.A	60	116	44	135	607	742
S681200.A	20	29	28	343	450	793
S662000.A	66	156	38	171	683	854
S686000.A	84	137	27	179	688	867
S682200.A	58	60	42	357	557	914
S682K00.A	52	64	40	381	547	928
S652000.A	60	148	34	170	792	962
S682300.A	54	49	44	402	589	991
S653000.A	70	180	34	217	960	1177
S622000.A	46	149	61	362	855	1217
S682500.A	68	73	47	500	732	1232
S687000.A	92	206	46	253	993	1246
S661000.A	86	224	43	262	1056	1318
S683001.A	78	152	79	431	974	1405
S621000.A	26	207	47	416	1002	1418
S621001.A	26	196	70	672	1615	2287
	19.02	29.97	26.16	73.25	163.19	236.44
	307	1522	114	9393	51783	100486
	18	39	11	97	228	317

TIME PREDICTED BY SCA	TIME ACTUAL (USER)	TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
2.51	1.2	1.0	2.2	0.31
2.51	1.2	1.1	2.3	0.21
2.48	0.8	1.0	1.8	0.68
2.76	0.8	1.1	1.9	0.86
2.76	1.5	1.6	3.1	-0.34
2.91	1.6	0.9	2.5	0.41
2.85	2.2	2.5	4.7	-1.85
2.85	1.5	2.0	3.5	-0.65
2.94	2.5	1.7	4.2	-1.26
3.01	1.5	1.6	3.1	-0.09
3.12	1.7	2.2	3.9	-0.78
3.13	0.9	0.9	1.8	1.33
3.13	0.9	0.9	1.8	1.33
3.40	2.5	1.9	4.4	-1.00
3.30	2.6	1.8	4.4	-1.10
3.43	0.9	1.0	1.9	1.53
3.28	1.6	1.7	3.3	-0.02
3.44	1.4	1.7	3.1	0.34
3.28	0.9	0.9	1.8	1.48
3.41	2.5	1.9	4.4	-0.99
3.52	1.6	2.1	3.7	-0.18
3.54	1.0	0.9	1.9	1.64
3.60	2.6	1.8	4.4	-0.80
3.68	0.9	0.9	1.8	1.88
3.72	1.3	1.2	2.5	1.22
3.67	1.4	1.2	2.6	1.07
3.83	0.9	0.9	1.8	2.03
3.87	2.1	1.8	3.9	-0.03
4.18	1.7	2.2	3.9	0.28
3.97	2.6	1.9	4.5	-0.53
4.07	2.7	1.9	4.6	-0.53
4.03	0.9	0.9	1.8	2.23
4.03	1.0	0.9	1.9	2.13
4.12	1.4	1.2	2.6	1.52
4.21	2.8	1.9	4.7	-0.49
4.21	2.6	2.0	4.6	-0.39
4.08	1.0	1.0	2.0	2.08
4.38	2.7	1.9	4.6	-0.22
4.37	2.2	1.2	3.4	0.97
4.32	1.9	1.9	3.8	0.52
4.14	1.0	1.0	2.0	2.14
4.25	0.9	1.0	1.9	2.35
4.25	1.0	1.0	2.0	2.25
4.35	1.4	1.1	2.5	1.85
4.41	2.0	1.8	3.8	0.61
4.50	2.8	1.8	4.6	-0.10

TIME PREDICTED BY SCA	TIME ACTUAL (USER)	TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
4.65	1.2	1.0	2.2	2.45
4.48	1.4	1.2	2.6	1.88
4.99	2.0	2.2	4.2	0.79
4.49	2.6	2.0	4.6	-0.11
4.67	2.2	2.2	4.4	0.27
4.66	0.9	0.9	1.8	2.86
4.93	1.1	1.0	2.1	2.83
4.89	1.5	1.6	3.1	1.79
4.65	1.0	0.9	1.9	2.75
4.72	1.9	1.8	3.7	1.02
5.01	1.4	1.1	2.5	2.51
4.90	2.9	2.0	4.9	0.00
5.07	1.3	1.1	2.4	2.67
5.03	1.5	1.1	2.6	2.43
5.28	1.1	1.0	2.1	3.18
5.36	3.7	2.6	6.3	-0.94
5.20	2.8	1.9	4.7	0.50
5.05	1.0	0.9	1.9	3.15
5.11	2.1	1.6	3.7	1.41
5.27	2.9	1.9	4.8	0.47
5.32	2.4	1.8	4.2	1.12
5.01	1.0	1.0	2.0	3.01
5.19	1.9	1.8	3.7	1.49
5.24	1.5	1.1	2.6	2.64
5.24	1.6	1.2	2.8	2.44
5.21	3.3	1.9	5.2	0.01
5.21	3.2	2.1	5.3	-0.09
5.80	2.6	2.5	5.1	0.70
5.36	2.1	1.8	3.9	1.46
5.49	2.4	1.7	4.1	1.39
5.71	1.3	1.1	2.4	3.31
5.71	1.3	1.0	2.3	3.41
5.50	2.0	1.6	3.6	1.90
5.64	2.5	1.9	4.4	1.24
5.61	1.7	1.1	2.8	2.81
5.71	1.4	1.2	2.6	3.11
6.26	2.8	2.6	5.4	0.86
5.93	1.0	1.0	2.0	3.93
5.76	3.6	2.0	5.6	0.16
5.90	2.8	1.9	4.7	1.20
5.85	1.3	1.1	2.4	3.45
6.52	1.7	2.3	4.0	2.52
6.07	1.5	1.1	2.6	3.47
6.23	2.9	1.9	4.8	1.43
6.42	1.3	1.3	2.6	3.82
6.32	2.2	1.7	3.9	2.42

TIME PREDICTED BY SCA	TIME ACTUAL (USER)	TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
6.27	1.7	1.2	2.9	3.37
6.21	1.7	1.2	2.9	3.31
6.26	1.1	1.0	2.1	4.16
5.81	1.0	0.9	1.9	3.91
6.42	3.4	2.0	5.4	1.02
7.15	2.8	2.5	5.3	1.85
6.23	1.4	1.2	2.6	3.63
6.67	1.5	1.1	2.6	4.07
6.60	3.3	1.9	5.2	1.40
6.67	1.8	1.2	3.0	3.67
6.82	1.0	1.1	2.1	4.72
6.79	1.5	1.1	2.6	4.19
6.79	3.6	2.0	5.6	1.19
6.94	1.2	1.0	2.2	4.74
6.78	1.1	1.1	2.2	4.58
7.46	2.4	2.5	4.9	2.56
6.86	2.6	1.6	4.2	2.66
6.83	3.8	2.0	5.8	1.03
7.53	2.1	2.4	4.5	3.03
7.67	1.0	0.9	1.9	5.77
6.93	3.0	2.0	5.0	1.93
7.22	3.8	2.1	5.9	1.32
8.10	4.2	1.4	5.6	2.50
7.33	2.6	1.8	4.4	2.93
6.97	1.3	1.0	2.3	4.67
8.41	3.0	2.1	5.1	3.31
7.04	1.3	1.2	2.5	4.54
7.67	4.2	2.0	6.2	1.47
8.52	1.5	1.2	2.7	5.82
8.61	3.0	2.5	5.5	3.11
7.83	2.0	1.4	3.4	4.43
8.90	1.7	1.1	2.8	6.10
7.78	2.8	1.8	4.6	3.18
8.09	4.4	2.0	6.4	1.69
9.12	2.1	2.2	4.3	4.82
8.20	1.6	1.1	2.7	5.50
8.27	1.6	1.1	2.7	5.57
8.71	1.4	1.0	2.4	6.31
8.84	2.5	1.6	4.1	4.74
8.55	1.3	1.0	2.3	6.25
8.86	1.5	1.0	2.5	6.36
8.41	1.7	1.1	2.8	5.61
8.92	1.5	1.1	2.6	6.32
8.58	3.7	2.2	5.9	2.68
8.65	1.5	1.3	2.8	5.85
8.10	1.2	0.9	2.1	6.00

TIME PREDICTED BY SCA	TIME ACTUAL (USER)	TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
8.69	4.2	2.1	6.3	2.39
9.08	1.5	1.1	2.6	6.48
8.69	1.9	1.2	3.1	5.59
8.59	1.7	1.0	2.7	5.89
9.31	4.8	2.2	7.0	2.31
9.28	1.4	1.0	2.4	6.88
8.89	3.9	2.1	6.0	2.89
8.03	1.3	1.1	2.4	5.63
8.53	1.3	1.0	2.3	6.23
9.17	4.7	2.1	6.8	2.37
9.12	4.6	1.9	6.5	2.62
8.96	1.7	1.1	2.8	6.16
10.59	1.7	1.0	2.7	7.89
9.21	4.5	1.8	6.3	2.91
10.32	2.2	1.1	3.3	7.02
9.53	1.7	1.0	2.7	6.83
9.80	1.9	1.1	3.0	6.80
9.80	1.7	1.2	2.9	6.90
8.82	1.4	1.0	2.4	6.42
9.48	1.3	1.0	2.3	7.18
9.71	1.5	1.1	2.6	7.11
12.04	2.9	2.2	5.1	6.94
11.27	4.7	1.5	6.2	5.07
11.72	1.1	0.8	1.9	9.82
13.15	1.1	1.0	2.1	11.05
10.13	1.7	1.2	2.9	7.23
12.01	1.0	1.0	2.0	10.01
11.88	2.3	1.0	3.3	8.58
11.99	4.7	1.5	6.2	5.79
12.49	5.2	1.8	7.0	5.49
12.21	3.7	1.5	5.2	7.01
13.25	2.2	1.1	3.3	9.95
11.59	2.3	1.1	3.4	8.19
13.69	4.5	2.7	7.2	6.49
11.66	1.9	1.1	3.0	8.66
10.53	1.4	1.0	2.4	8.13
12.96	2.9	2.4	5.3	7.66
10.54	1.6	1.1	2.7	7.84
12.38	2.2	1.2	3.4	8.98
13.68	4.4	1.8	6.2	7.48
13.37	2.3	1.3	3.6	9.77
14.90	2.5	1.1	3.6	11.30
15.75	3.6	1.2	4.8	10.95
16.21	5.3	2.7	8.0	8.21
14.71	1.5	0.9	2.4	12.31
15.33	3.8	1.3	5.1	10.23

	TIME PREDICTED BY SCA	TIME ACTUAL (USER)	TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
	15.00	2.8	1.3	4.1	10.90
	18.09	5.0	2.7	7.7	10.39
	15.41	2.1	1.2	3.3	12.11
	18.88	6.4	3.0	9.4	9.48
	15.20	2.3	1.2	3.5	11.70
	20.90	6.5	3.0	9.5	11.40
	20.98	7.8	3.0	10.8	10.18
	17.74	3.2	1.2	4.4	13.34
	17.98	3.0	1.3	4.3	13.68
	22.11	5.7	2.7	8.4	13.71
	17.78	3.2	1.2	4.4	13.38
	24.65	6.5	2.8	9.3	15.35
	22.73	4.0	1.5	5.5	17.23
	20.46	3.7	1.2	4.9	15.56
	25.11	9.2	3.0	12.2	12.91
	26.15	7.9	2.9	10.8	15.35
	23.83	4.3	1.3	5.6	18.23
	25.74	6.3	1.4	7.7	18.04
	30.39	2.9	1.1	4.0	26.39
MEAN	8.24	2.36	1.52	3.88	4.35
VARIANCE	27.02	2.04	0.31	3.50	19.51
STND DEV	5.20	1.43	0.55	1.87	4.42

STATISTICS FOR ORIGINAL MODEL

MEAN PREDICTION ERROR = 4.35

VARIANCE PREDICTION ERROR = 19.51

STANDARD DEVIATION PREDICTION ERROR = 4.42

Appendix A2: Linear Regression

	FILE					FILE
CAMP	SIZE					LENGTH
FILENAME	(K)	n2	n1	N2	N1	N
S681240.A	4	3	12	4	14	18
S681230.A	4	3	12	4	14	18
S644001.A	6	3	7	4	15	19
S002B00.A	6	3	15	5	17	22
S001700.A	6	3	15	5	17	22
S682W00.A	12	6	8	7	16	23
S662001.A	4	4	7	5	19	24
S651001.A	4	4	7	5	19	24
S671500.A	6	4	16	6	18	24
S001800.A	8	4	15	7	19	26
S361001.A	8	4	15	7	21	28
S002D00.A	8	4	16	7	21	28
S002E00.A	8	4	16	7	21	28
S671300.A	6	6	18	9	22	31
S671400.A	6	5	15	9	22	31
S002800.A	8	6	16	9	23	32
S001200.A	8	4	16	9	23	32
S001500.A	8	6	17	9	23	32
S002400.A	8	4	16	9	23	32
S671200.A	6	5	16	9	24	33
S361000.A	8	5	16	6	27	33
S644121.A	8	6	16	9	25	34
S671100.A	8	7	17	11	24	35
S002J00.A	6	7	17	10	26	36
S687E00.A	6	6	18	12	27	39
S687D00.A	6	5	16	12	28	40
S002K00.A	8	7	18	12	28	40
S001400.A	10	8	18	13	27	40
S615000.A	6	12	9	12	30	42
S661600.A	10	8	20	13	29	42
S661A00.A	10	8	21	13	31	44
S002F00.A	8	7	16	14	32	46
S002G00.A	8	7	16	14	32	46
S687F00.A	8	8	22	15	31	46
S661810.A	8	8	20	14	34	48
S661820.A	8	8	20	14	34	48
S002C00.A	10	6	20	14	34	48
S661800.A	6	10	15	11	37	48
S682Z00.A	30	11	11	17	33	50

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	FILE LENGTH N
S001300.A	12	9	22	16	34	50
S002500.A	10	6	20	15	35	50
S002100.A	8	7	20	15	36	51
S002200.A	10	7	20	15	36	51
S682S00.A	8	10	19	19	32	51
S652200.A	8	7	24	16	39	55
S684200.A	8	9	21	17	38	55
S682Q00.A	8	13	19	22	34	56
S687A00.A	8	7	19	15	42	57
S613000.A	6	17	11	17	40	57
S661700.A	8	8	17	17	40	57
S653200.A	8	9	24	17	42	59
S002I00.A	8	9	21	19	41	60
S644240.A	12	15	24	21	39	60
S634000.A	10	12	17	19	43	62
S002H00.A	8	8	18	19	43	62
S652300.A	8	8	24	19	44	63
S682V00.A	10	14	23	25	40	65
S662100.A	20	8	25	16	50	66
S682P00.A	8	14	23	26	41	67
S687C00.A	8	11	31	21	46	67
S644120.A	16	14	21	15	52	67
S686001.A	8	12	12	13	58	71
S684300.A	10	12	23	23	49	72
S002300.A	10	9	22	23	50	73
S651300.A	20	8	23	17	56	73
S684500.A	10	12	24	23	51	74
S653300.A	10	12	29	23	52	75
S002A00.A	10	9	20	30	45	75
S652100.A	8	10	26	24	51	75
S687300.A	14	10	19	21	55	76
S687400.A	14	10	19	21	55	76
S686200.A	14	9	21	21	56	77
S686300.A	14	9	21	21	56	77
S671001.A	6	16	16	17	63	80
S652600.A	10	10	27	24	56	80
S653600.A	10	11	29	24	58	82
S682N00.A	8	18	23	33	50	83
S682R00.A	8	18	23	33	50	83
S651200.A	22	9	23	20	65	85
S653100.A	10	13	28	29	57	86
S687800.A	16	10	19	23	66	89
S687G00.A	8	13	26	32	58	90
S684001.A	6	18	17	19	73	92
S002900.A	12	15	23	32	62	94
S686400.A	16	10	24	25	69	94

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	FILE LENGTH N
S684100.A	10	15	21	34	60	94
S682T00.A	8	13	25	38	59	97
S001001.A	8	20	21	21	78	99
S682L00.A	16	15	22	33	66	99
S661400.A	12	16	31	32	70	102
S661320.A	12	19	24	34	71	105
S001100.A	22	15	23	32	75	107
S687500.A	16	13	21	31	78	109
S687600.A	18	11	20	28	82	110
S002600.A	12	14	26	39	71	110
S644170.A	10	9	20	49	61	110
S686100.A	18	13	27	35	81	116
S681001.A	8	23	19	24	93	117
S681500.A	6	12	22	47	70	117
S682E00.A	14	19	31	44	74	118
S671600.A	16	16	27	40	79	119
S687900.A	20	16	23	34	85	119
S622001.A	8	20	23	42	79	121
S682M00.A	18	20	23	45	76	121
S661520.A	14	18	32	44	80	124
S644230.A	16	21	26	47	79	126
S002700.A	14	18	21	49	77	126
S653001.A	6	25	19	26	100	126
S001600.A	32	16	26	36	91	127
S686500.A	20	15	25	36	92	128
S652001.A	6	25	21	26	103	129
S644100.A	8	27	23	28	105	133
S661530.A	14	14	28	46	93	139
S686800.A	24	17	23	43	100	143
S687001.A	6	29	19	30	116	146
S651100.A	32	16	32	39	108	147
S644150.A	16	14	20	67	83	150
S661500.A	14	31	21	32	125	157
S681600.A	6	12	23	68	91	159
S686700.A	24	19	24	49	111	160
S632000.A	12	39	26	46	115	161
S672000.A	14	34	19	36	126	162
S687700.A	26	16	21	47	127	174
S631000.A	14	42	26	49	125	174
S634001.A	36	16	20	53	122	175
S686600.A	24	21	24	55	122	177
S602000.A	14	40	23	42	138	180
S682C00.A	18	26	35	71	109	180
S682U00.A	12	27	38	72	110	182
S682X00.A	12	35	30	61	122	183
S661300.A	14	35	27	52	131	183

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	FILE LENGTH N
S644220.A	20	31	31	63	120	183
S611000.A	10	34	21	62	126	188
S682B00.A	18	27	36	76	114	190
S612000.A	10	34	21	62	129	191
S652500.A	22	29	37	70	122	192
S661310.A	14	27	27	65	130	195
S644210.A	12	18	25	79	117	196
S653500.A	24	30	37	72	124	196
S682Y00.A	14	35	31	66	136	202
S682G00.A	30	25	25	70	133	203
S682900.A	18	24	31	74	129	203
S684400.A	18	40	39	70	139	209
S644200.A	24	36	37	72	142	214
S652400.A	20	26	31	82	135	217
S644160.A	16	13	21	101	117	218
S644110.A	18	17	29	92	134	226
S661510.A	20	28	38	80	147	227
S653400.A	22	27	31	88	141	229
S682700.A	22	24	33	91	138	229
S682001.A	18	48	23	49	188	237
S686900.A	26	24	27	80	157	237
S614000.A	12	45	15	81	158	239
S682F00.A	18	29	36	106	148	254
S682A00.A	20	32	38	110	154	264
S682D00.A	20	32	38	110	154	264
S644130.A	16	14	25	118	147	265
S644180.A	16	19	23	123	163	286
S644122.A	22	18	27	134	175	309
S651000.A	30	51	24	65	253	318
S686A00.A	44	44	48	108	218	326
S632001.A	6	48	38	116	229	345
S002001.A	14	74	25	75	273	348
S681400.A	12	17	26	151	198	349
S631001.A	6	50	38	120	241	361
S682H00.A	38	45	38	133	236	369
S686B00.A	34	48	50	127	245	372
S662300.A	52	58	46	119	260	379
S661900.A	36	48	35	131	250	381
S683000.A	22	65	30	84	297	381
S682100.A	38	33	36	141	243	384
S671000.A	42	63	23	71	321	392
S682800.A	24	32	36	173	232	405
S644140.A	14	14	26	182	226	408
S602001.A	14	52	38	132	292	424
S681700.A	8	12	23	206	229	435
S687200.A	24	34	37	177	276	453

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	FILE LENGTH N
S672001.A	56	61	37	146	313	459
S687100.A	28	47	43	181	310	491
S623000.A	22	80	48	136	375	511
S688000.A	80	85	36	106	423	529
S684000.A	46	89	31	104	440	544
S623001.A	12	52	52	146	423	569
S682600.A	80	71	47	195	397	592
S682J00.A	44	45	40	258	406	664
S001000.A	64	111	36	126	541	667
S687H00.A	16	41	44	287	447	734
S681000.A	60	116	44	135	607	742
S681200.A	20	29	28	343	450	793
S662000.A	66	156	38	171	683	854
S686000.A	84	137	27	179	688	867
S682200.A	58	60	42	357	557	914
S682K00.A	52	64	40	381	547	928
S652000.A	60	148	34	170	792	962
S682300.A	54	49	44	402	589	991
S653000.A	70	180	34	217	960	1177
S622000.A	46	149	61	362	855	1217
S682500.A	68	73	47	500	732	1232
S687000.A	92	206	46	253	993	1246
S661000.A	86	224	43	262	1056	1318
S683001.A	78	152	79	431	974	1405
S621000.A	26	207	47	416	1002	1418
S621001.A	26	196	70	672	1615	2287
MEAN	19	30	26	73	163	236
VARIANCE	307	1522	114	9393	51783	100486
STND DEV	18	39	11	97	228	317

VOCAB n	VOLUME V	POTEN VOL V* (L^)	T = TIME PREDICTED BY SCA	TIME ACTUAL (USER)
15	70.32	0.125	2.51	1.2
15	70.32	0.125	2.51	1.2
10	63.12	0.214	2.48	0.8
18	91.74	0.080	2.76	0.8
18	91.74	0.080	2.76	1.5
14	87.57	0.214	2.91	1.6
11	83.03	0.229	2.85	2.2
11	83.03	0.229	2.85	1.5
20	103.73	0.083	2.94	2.5
19	110.45	0.076	3.01	1.5
19	118.94	0.076	3.12	1.7
20	121.01	0.071	3.13	0.9
20	121.01	0.071	3.13	0.9
24	142.13	0.074	3.40	2.5
20	133.98	0.074	3.30	2.6
22	142.70	0.083	3.43	0.9
20	138.30	0.056	3.28	1.6
23	144.75	0.078	3.44	1.4
20	138.30	0.056	3.28	0.9
21	144.95	0.069	3.41	2.5
21	144.95	0.104	3.52	1.6
22	151.62	0.083	3.54	1.0
24	160.47	0.075	3.60	2.6
24	165.06	0.082	3.68	0.9
24	178.81	0.056	3.72	1.3
21	175.69	0.052	3.67	1.4
25	185.75	0.065	3.83	0.9
26	188.02	0.068	3.87	2.1
21	184.48	0.222	4.18	1.7
28	201.91	0.062	3.97	2.6
29	213.75	0.059	4.07	2.7
23	208.08	0.063	4.03	0.9
23	208.08	0.063	4.03	1.0
30	225.72	0.048	4.12	1.4
28	230.75	0.057	4.21	2.8
28	230.75	0.057	4.21	2.6
26	225.62	0.043	4.08	1.0
25	222.91	0.121	4.38	2.7
22	222.97	0.118	4.37	2.2
31	247.71	0.051	4.32	1.9
26	235.02	0.040	4.14	1.0
27	242.50	0.047	4.25	0.9
27	242.50	0.047	4.25	1.0
29	247.76	0.055	4.35	1.4
31	272.48	0.036	4.41	2.0
30	269.88	0.050	4.50	2.8

VOCAB n	VOLUME V	POTEN VOL V* (L^)	T = TIME PREDICTED BY SCA	TIME ACTUAL (USER)
32	280.00	0.062	4.65	1.2
26	267.93	0.049	4.48	1.4
28	274.02	0.182	4.99	2.0
25	264.70	0.055	4.49	2.6
33	297.62	0.044	4.67	2.2
30	294.41	0.045	4.66	0.9
39	317.12	0.060	4.93	1.1
29	301.19	0.074	4.89	1.5
26	291.43	0.047	4.65	1.0
32	315.00	0.035	4.72	1.9
37	338.61	0.049	5.01	1.4
33	332.93	0.040	4.90	2.9
37	349.03	0.047	5.07	1.3
42	361.29	0.034	5.03	1.5
35	343.66	0.089	5.28	1.1
24	325.53	0.154	5.36	3.7
35	369.31	0.045	5.20	2.8
31	361.66	0.036	5.05	1.0
31	361.66	0.041	5.11	2.1
36	382.57	0.043	5.27	2.9
41	401.82	0.036	5.32	2.4
29	364.35	0.030	5.01	1.0
36	387.74	0.032	5.19	1.9
29	369.21	0.050	5.24	1.5
29	369.21	0.050	5.24	1.6
30	377.83	0.041	5.21	3.3
30	377.83	0.041	5.21	3.2
32	400.00	0.118	5.80	2.6
37	416.76	0.031	5.36	2.1
40	436.40	0.032	5.49	2.4
41	444.68	0.047	5.71	1.3
41	444.68	0.047	5.71	1.3
32	425.00	0.039	5.50	2.0
41	460.75	0.032	5.64	2.5
29	432.36	0.046	5.61	1.7
39	475.69	0.031	5.71	1.4
35	471.89	0.111	6.26	2.8
38	493.31	0.041	5.93	1.0
34	478.22	0.033	5.76	3.6
36	485.97	0.042	5.90	2.8
38	509.05	0.027	5.85	1.3
41	530.40	0.091	6.52	1.7
37	515.74	0.041	6.07	1.5
47	566.57	0.032	6.23	2.9
43	569.76	0.047	6.42	1.3
38	561.53	0.041	6.32	2.2

VOCAB n	VOLUME V	POTEN VOL V* (L^)	T = TIME PREDICTED BY SCA	TIME ACTUAL (USER)
34	554.53	0.040	6.27	1.7
31	544.96	0.039	6.21	1.7
40	585.41	0.028	6.26	1.1
29	534.38	0.018	5.81	1.0
40	617.34	0.028	6.42	3.4
42	630.90	0.101	7.15	2.8
34	595.23	0.023	6.23	1.4
50	665.98	0.028	6.67	1.5
43	645.73	0.030	6.60	3.3
39	628.96	0.041	6.67	1.8
43	656.58	0.041	6.82	1.0
43	656.58	0.039	6.79	1.5
50	699.84	0.026	6.79	3.6
47	699.88	0.034	6.94	1.2
39	665.96	0.035	6.78	1.1
44	687.89	0.101	7.46	2.4
42	684.82	0.034	6.86	2.6
40	681.21	0.033	6.83	3.8
46	712.54	0.092	7.53	2.1
50	750.63	0.084	7.67	1.0
42	749.53	0.022	6.93	3.0
40	761.04	0.034	7.22	3.8
48	815.40	0.102	8.10	4.2
48	820.99	0.026	7.33	2.6
34	763.12	0.021	6.97	1.3
52	894.97	0.092	8.41	3.0
35	815.56	0.015	7.04	1.3
43	868.20	0.032	7.67	4.2
65	969.60	0.065	8.52	1.5
53	927.92	0.099	8.61	3.0
37	906.44	0.032	7.83	2.0
68	1059.22	0.066	8.90	1.7
36	904.74	0.030	7.78	2.8
45	972.06	0.032	8.09	4.4
63	1075.91	0.083	9.12	2.1
61	1067.53	0.021	8.20	1.6
65	1096.07	0.020	8.27	1.6
65	1102.09	0.038	8.71	1.4
62	1089.62	0.050	8.84	2.5
62	1089.62	0.032	8.55	1.3
55	1086.90	0.052	8.86	1.5
63	1135.68	0.020	8.41	1.7
55	1104.24	0.052	8.92	1.5
66	1160.52	0.022	8.58	3.7
54	1122.20	0.031	8.65	1.5
43	1063.55	0.018	8.10	1.2

VOCAB n	VOLUME V	POTEN VOL V* (L^)	T = TIME PREDICTED BY SCA	TIME ACTUAL (USER)
67	1188.95	0.023	8.69	4.2
66	1220.97	0.034	9.08	1.5
50	1145.70	0.029	8.69	1.9
55	1173.62	0.021	8.59	1.7
79	1317.49	0.029	9.31	4.8
73	1324.62	0.027	9.28	1.4
57	1265.74	0.020	8.89	3.9
34	1109.07	0.012	8.03	1.3
46	1248.33	0.013	8.53	1.3
66	1372.08	0.018	9.17	4.7
58	1341.48	0.020	9.12	4.6
57	1335.73	0.016	8.96	1.7
71	1457.49	0.085	10.59	1.7
51	1344.36	0.022	9.21	4.5
60	1411.75	0.074	10.32	2.2
65	1529.68	0.015	9.53	1.7
70	1618.13	0.015	9.80	1.9
70	1618.13	0.015	9.80	1.7
39	1400.63	0.009	8.82	1.1
42	1542.20	0.013	9.48	1.3
45	1696.98	0.010	9.71	1.5
75	1980.76	0.065	12.04	2.9
92	2126.68	0.017	11.27	4.7
86	2217.06	0.022	11.72	1.1
99	2307.02	0.079	13.15	1.1
43	1893.77	0.009	10.13	1.7
88	2331.86	0.022	12.01	1.0
83	2352.39	0.018	11.88	2.3
98	2460.67	0.015	11.99	4.7
104	2539.47	0.021	12.49	5.2
83	2428.89	0.021	12.21	3.7
95	2503.12	0.052	13.25	2.2
69	2345.67	0.013	11.59	2.3
86	2519.10	0.077	13.69	4.5
68	2465.42	0.010	11.66	1.9
40	2171.35	0.006	10.53	1.4
90	2752.55	0.021	12.96	2.9
35	2231.24	0.005	10.54	1.6
71	2785.84	0.010	12.38	2.2
98	3036.15	0.023	13.68	4.4
90	3187.50	0.012	13.37	2.3
128	3577.00	0.025	14.90	2.5
121	3660.08	0.045	15.75	3.6
120	3757.35	0.055	16.21	5.3
104	3812.55	0.014	14.71	1.5
118	4074.53	0.015	15.33	3.8

VOCAB n	VOLUME V	POTEN VOL V* (L^)	T = TIME PREDICTED BY SCA	TIME ACTUAL (USER)
85	4255.84	0.009	15.00	2.8
147	4802.18	0.049	18.09	5.0
85	4704.49	0.006	15.41	2.1
160	5432.87	0.039	18.88	6.4
57	4625.48	0.006	15.20	2.3
194	6490.33	0.048	20.90	6.5
164	6379.00	0.057	20.98	7.8
102	6098.60	0.008	17.74	3.2
104	6218.01	0.008	17.98	3.0
182	7222.50	0.051	22.11	5.7
93	6480.31	0.006	17.78	3.2
214	9111.71	0.049	24.65	6.5
210	9388.24	0.013	22.73	4.0
120	8509.29	0.006	20.46	3.7
252	9939.69	0.035	25.11	9.2
267	10624.00	0.040	26.15	7.9
231	11031.71	0.009	23.83	4.3
254	11327.96	0.021	25.74	6.3
266	18422.43	0.008	30.39	2.9
56	1534.23	0.050	8.24	2.36
2210	5950085.02	0.002	27.02	2.04
47	2439.28	0.040	5.20	1.43

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL	Ln(V) INDEPENDENT VARIABLE	Ln(L^) INDEPENDENT VARIABLE
1.0	2.2	0.31	4.25	-2.08
1.1	2.3	0.21	4.25	-2.08
1.0	1.8	0.68	4.14	-1.54
1.1	1.9	0.86	4.52	-2.53
1.6	3.1	-0.34	4.52	-2.53
0.9	2.5	0.41	4.47	-1.54
2.5	4.7	-1.85	4.42	-1.48
2.0	3.5	-0.65	4.42	-1.48
1.7	4.2	-1.26	4.64	-2.48
1.6	3.1	-0.09	4.70	-2.57
2.2	3.9	-0.78	4.78	-2.57
0.9	1.8	1.33	4.80	-2.64
0.9	1.8	1.33	4.80	-2.64
1.9	4.4	-1.00	4.96	-2.60
1.8	4.4	-1.10	4.90	-2.60
1.0	1.9	1.53	4.96	-2.48
1.7	3.3	-0.02	4.93	-2.89
1.7	3.1	0.34	4.98	-2.55
0.9	1.8	1.48	4.93	-2.89
1.9	4.4	-0.99	4.98	-2.67
2.1	3.7	-0.18	4.98	-2.26
0.9	1.9	1.64	5.02	-2.48
1.8	4.4	-0.80	5.08	-2.59
0.9	1.8	1.88	5.11	-2.50
1.2	2.5	1.22	5.19	-2.89
1.2	2.6	1.07	5.17	-2.95
0.9	1.8	2.03	5.22	-2.74
1.8	3.9	-0.03	5.24	-2.68
2.2	3.9	0.28	5.22	-1.50
1.9	4.5	-0.53	5.31	-2.79
1.9	4.6	-0.53	5.36	-2.84
0.9	1.8	2.23	5.34	-2.77
0.9	1.9	2.13	5.34	-2.77
1.2	2.6	1.52	5.42	-3.03
1.9	4.7	-0.49	5.44	-2.86
2.0	4.6	-0.39	5.44	-2.86
1.0	2.0	2.08	5.42	-3.15
1.9	4.6	-0.22	5.41	-2.11
1.2	3.4	0.97	5.41	-2.14
1.9	3.8	0.52	5.51	-2.97
1.0	2.0	2.14	5.46	-3.22
1.0	1.9	2.35	5.49	-3.06
1.0	2.0	2.25	5.49	-3.06
1.1	2.5	1.85	5.51	-2.89
1.8	3.8	0.61	5.61	-3.31
1.8	4.6	-0.10	5.60	-2.99

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL	Ln(V) INDEPENDENT VARIABLE	Ln(L^) INDEPENDENT VARIABLE
1.0	2.2	2.45	5.63	-2.78
1.2	2.6	1.88	5.59	-3.01
2.2	4.2	0.79	5.61	-1.70
2.0	4.6	-0.11	5.58	-2.89
2.2	4.4	0.27	5.70	-3.12
0.9	1.8	2.86	5.68	-3.10
1.0	2.1	2.83	5.76	-2.82
1.6	3.1	1.79	5.71	-2.60
0.9	1.9	2.75	5.67	-3.06
1.8	3.7	1.02	5.75	-3.35
1.1	2.5	2.51	5.82	-3.02
2.0	4.9	0.00	5.81	-3.22
1.1	2.4	2.67	5.86	-3.06
1.1	2.6	2.43	5.89	-3.39
1.0	2.1	3.18	5.84	-2.42
2.6	6.3	-0.94	5.79	-1.87
1.9	4.7	0.50	5.91	-3.09
0.9	1.9	3.15	5.89	-3.34
1.6	3.7	1.41	5.89	-3.20
1.9	4.8	0.47	5.95	-3.14
1.8	4.2	1.12	6.00	-3.32
1.0	2.0	3.01	5.90	-3.51
1.8	3.7	1.49	5.96	-3.44
1.1	2.6	2.64	5.91	-2.99
1.2	2.8	2.44	5.91	-2.99
1.9	5.2	0.01	5.93	-3.20
2.1	5.3	-0.09	5.93	-3.20
2.5	5.1	0.70	5.99	-2.14
1.8	3.9	1.46	6.03	-3.48
1.7	4.1	1.39	6.08	-3.45
1.1	2.4	3.31	6.10	-3.05
1.0	2.3	3.41	6.10	-3.05
1.6	3.6	1.90	6.05	-3.24
1.9	4.4	1.24	6.13	-3.44
1.1	2.8	2.81	6.07	-3.08
1.2	2.6	3.11	6.16	-3.47
2.6	5.4	0.86	6.16	-2.19
1.0	2.0	3.93	6.20	-3.20
2.0	5.6	0.16	6.17	-3.40
1.9	4.7	1.20	6.19	-3.17
1.1	2.4	3.45	6.23	-3.60
2.3	4.0	2.52	6.27	-2.40
1.1	2.6	3.47	6.25	-3.19
1.9	4.8	1.43	6.34	-3.43
1.3	2.6	3.82	6.35	-3.07
1.7	3.9	2.42	6.33	-3.20

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL	Ln(V) INDEPENDENT VARIABLE	Ln(L^) INDEPENDENT VARIABLE
1.2	2.9	3.37	6.32	-3.22
1.2	2.9	3.31	6.30	-3.24
1.0	2.1	4.16	6.37	-3.59
0.9	1.9	3.91	6.28	-4.00
2.0	5.4	1.02	6.43	-3.59
2.5	5.3	1.85	6.45	-2.29
1.2	2.6	3.63	6.39	-3.76
1.1	2.6	4.07	6.50	-3.58
1.9	5.2	1.40	6.47	-3.52
1.2	3.0	3.67	6.44	-3.20
1.1	2.1	4.72	6.49	-3.18
1.1	2.6	4.19	6.49	-3.25
2.0	5.6	1.19	6.55	-3.67
1.0	2.2	4.74	6.55	-3.37
1.1	2.2	4.58	6.50	-3.35
2.5	4.9	2.56	6.53	-2.29
1.6	4.2	2.66	6.53	-3.38
2.0	5.8	1.03	6.52	-3.40
2.4	4.5	3.03	6.57	-2.39
0.9	1.9	5.77	6.62	-2.48
2.0	5.0	1.93	6.62	-3.83
2.1	5.9	1.32	6.63	-3.37
1.4	5.6	2.50	6.70	-2.29
1.8	4.4	2.93	6.71	-3.66
1.0	2.3	4.67	6.64	-3.87
2.1	5.1	3.31	6.80	-2.38
1.2	2.5	4.54	6.70	-4.18
2.0	6.2	1.47	6.77	-3.43
1.2	2.7	5.82	6.88	-2.73
2.5	5.5	3.11	6.83	-2.31
1.4	3.4	4.43	6.81	-3.43
1.1	2.8	6.10	6.97	-2.72
1.8	4.6	3.18	6.81	-3.50
2.0	6.4	1.69	6.88	-3.45
2.2	4.3	4.82	6.98	-2.49
1.1	2.7	5.50	6.97	-3.87
1.1	2.7	5.57	7.00	-3.93
1.0	2.4	6.31	7.00	-3.26
1.6	4.1	4.74	6.99	-3.00
1.0	2.3	6.25	6.99	-3.45
1.0	2.5	6.36	6.99	-2.95
1.1	2.8	5.61	7.03	-3.93
1.1	2.6	6.32	7.01	-2.95
2.2	5.9	2.68	7.06	-3.80
1.3	2.8	5.85	7.02	-3.48
0.9	2.1	6.00	6.97	-4.00

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL	Ln(V) INDEPENDENT VARIABLE	Ln(L^) INDEPENDENT VARIABLE
2.1	6.3	2.39	7.08	-3.79
1.1	2.6	6.48	7.11	-3.38
1.2	3.1	5.59	7.04	-3.56
1.0	2.7	5.89	7.07	-3.87
2.2	7.0	2.31	7.18	-3.53
1.0	2.4	6.88	7.19	-3.61
2.1	6.0	2.89	7.14	-3.89
1.1	2.4	5.63	7.01	-4.40
1.0	2.3	6.23	7.13	-4.36
2.1	6.8	2.37	7.22	-3.99
1.9	6.5	2.62	7.20	-3.92
1.1	2.8	6.16	7.20	-4.14
1.0	2.7	7.89	7.28	-2.46
1.8	6.3	2.91	7.20	-3.81
1.1	3.3	7.02	7.25	-2.60
1.0	2.7	6.83	7.33	-4.19
1.1	3.0	6.80	7.39	-4.18
1.2	2.9	6.90	7.39	-4.18
1.0	2.4	6.42	7.24	-4.66
1.0	2.3	7.18	7.34	-4.31
1.1	2.6	7.11	7.44	-4.61
2.2	5.1	6.94	7.59	-2.73
1.5	6.2	5.07	7.66	-4.08
0.8	1.9	9.82	7.70	-3.83
1.0	2.1	11.05	7.74	-2.54
1.2	2.9	7.23	7.55	-4.75
1.0	2.0	10.01	7.75	-3.82
1.0	3.3	8.58	7.76	-4.03
1.5	6.2	5.79	7.81	-4.19
1.8	7.0	5.49	7.84	-3.85
1.5	5.2	7.01	7.80	-3.87
1.1	3.3	9.95	7.83	-2.96
1.1	3.4	8.19	7.76	-4.34
2.7	7.2	6.49	7.83	-2.56
1.1	3.0	8.66	7.81	-4.58
1.0	2.4	8.13	7.68	-5.13
2.4	5.3	7.66	7.92	-3.88
1.1	2.7	7.84	7.71	-5.29
1.2	3.4	8.98	7.93	-4.57
1.8	6.2	7.48	8.02	-3.79
1.3	3.6	9.77	8.07	-4.42
1.1	3.6	11.30	8.18	-3.71
1.2	4.8	10.95	8.21	-3.11
2.7	8.0	8.21	8.23	-2.90
0.9	2.4	12.31	8.25	-4.29
1.3	5.1	10.23	8.31	-4.17

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL	Ln(V) INDEPENDENT VARIABLE	Ln(L^) INDEPENDENT VARIABLE
1.3	4.1	10.90	8.36	-4.74
2.7	7.7	10.39	8.48	-3.02
1.2	3.3	12.11	8.46	-5.04
3.0	9.4	9.48	8.60	-3.24
1.2	3.5	11.70	8.44	-5.11
3.0	9.5	11.40	8.78	-3.04
3.0	10.8	10.18	8.76	-2.87
1.2	4.4	13.34	8.72	-4.83
1.3	4.3	13.68	8.74	-4.78
2.7	8.4	13.71	8.88	-2.97
1.2	4.4	13.38	8.78	-5.20
2.8	9.3	15.35	9.12	-3.02
1.5	5.5	17.23	9.15	-4.31
1.2	4.9	15.56	9.05	-5.08
3.0	12.2	12.91	9.20	-3.34
2.9	10.8	15.35	9.27	-3.22
1.3	5.6	18.23	9.31	-4.72
1.4	7.7	18.04	9.34	-3.85
1.1	4.0	26.39	9.82	-4.79
	1.52	3.88	4.35	
	0.31	3.50	19.51	
	0.55	1.87	4.42	

Ln(T) DEPENDENT VARIABLE
0.79
0.83
0.59
0.64
1.13
0.92
1.55
1.25
1.44
1.13
1.36
0.59
0.59
1.48
1.48
0.64
1.19
1.13
0.59
1.48
1.31
0.64
1.48
0.59
0.92
0.96
0.59
1.36
1.36
1.50
1.53
0.59
0.64
0.96
1.55
1.53
0.69
1.53
1.22
1.34
0.69
0.64
0.69
0.92
1.34
1.53

Ln(T) DEPENDENT VARIABLE
0.79
0.96
1.44
1.53
1.48
0.59
0.74
1.13
0.64
1.31
0.92
1.59
0.88
0.96
0.74
1.84
1.55
0.64
1.31
1.57
1.44
0.69
1.31
0.96
1.03
1.65
1.67
1.63
1.36
1.41
0.88
0.83
1.28
1.48
1.03
0.96
1.69
0.69
1.72
1.55
0.88
1.39
0.96
1.57
0.96
1.36

Ln(T) DEPENDENT VARIABLE
1.06
1.06
0.74
0.64
1.69
1.67
0.96
0.96
1.65
1.10
0.74
0.96
1.72
0.79
0.79
1.59
1.44
1.76
1.50
0.64
1.61
1.77
1.72
1.48
0.83
1.63
0.92
1.82
0.99
1.70
1.22
1.03
1.53
1.86
1.46
0.99
0.99
0.88
1.41
0.83
0.92
1.03
0.96
1.77
1.03
0.74

Ln(T) DEPENDENT VARIABLE
1.84
0.96
1.13
0.99
1.95
0.88
1.79
0.88
0.83
1.92
1.87
1.03
0.99
1.84
1.19
0.99
1.10
1.06
0.88
0.83
0.96
1.63
1.82
0.64
0.74
1.06
0.69
1.19
1.82
1.95
1.65
1.19
1.22
1.97
1.10
0.88
1.67
0.99
1.22
1.82
1.28
1.28
1.57
2.08
0.88
1.63

```
   Ln(T)
 DEPENDENT
  VARIABLE
-----------
       1.41
       2.04
       1.19
       2.24
       1.25
       2.25
       2.38
       1.48
       1.46
       2.13
       1.48
       2.23
       1.70
       1.59
       2.50
       2.38
       1.72
       2.04
       1.39

                 Linear
            Regression Output:
Constant                                     0.3522
Std Err of Y Est                             0.3650
R Squared                                    0.3083
No. of Observations                        203.0000
Degrees of Freedom                         200.0000
X Coefficient(s)             0.2683          0.2609
Std Err of Coef.             0.0285          0.0448
```

Appendix A3: Analysis of Recalibrated Compile Time Model

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	LENGTH N
S681240.A	4	3	12	4	14	18
S681230.A	4	3	12	4	14	18
S644001.A	6	3	7	4	15	19
S002B00.A	6	3	15	5	17	22
S001700.A	6	3	15	5	17	22
S682W00.A	12	6	8	7	16	23
S662001.A	4	4	7	5	19	24
S651001.A	4	4	7	5	19	24
S671500.A	6	4	16	6	18	24
S001800.A	8	4	15	7	19	26
S361001.A	8	4	15	7	21	28
S002D00.A	8	4	16	7	21	28
S002E00.A	8	4	16	7	21	28
S671300.A	6	6	18	9	22	31
S671400.A	6	5	15	9	22	31
S002800.A	8	6	16	9	23	32
S001200.A	8	4	16	9	23	32
S001500.A	8	6	17	9	23	32
S002400.A	8	4	16	9	23	32
S671200.A	6	5	16	9	24	33
S361000.A	8	5	16	6	27	33
S644121.A	8	6	16	9	25	34
S671100.A	8	7	17	11	24	35
S002J00.A	6	7	17	10	26	36
S687E00.A	6	6	18	12	27	39
S687D00.A	6	5	16	12	28	40
S002K00.A	8	7	18	12	28	40
S001400.A	10	8	18	13	27	40
S615000.A	6	12	9	12	30	42
S661600.A	10	8	20	13	29	42
S661A00.A	10	8	21	13	31	44
S002F00.A	8	7	16	14	32	46
S002G00.A	8	7	16	14	32	46
S687F00.A	8	8	22	15	31	46
S661810.A	8	8	20	14	34	48
S661820.A	8	8	20	14	34	48
S002C00.A	10	6	20	14	34	48
S661800.A	6	10	15	11	37	48
S682Z00.A	30	11	11	17	33	50

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	LENGTH N
S001300.A	12	9	22	16	34	50
S002500.A	10	6	20	15	35	50
S002100.A	8	7	20	15	36	51
S002200.A	10	7	20	15	36	51
S682S00.A	8	10	19	19	32	51
S652200.A	8	7	24	16	39	55
S684200.A	8	9	21	17	38	55
S682Q00.A	8	13	19	22	34	56
S687A00.A	8	7	19	15	42	57
S613000.A	6	17	11	17	40	57
S661700.A	8	8	17	17	40	57
S653200.A	8	9	24	17	42	59
S002I00.A	8	9	21	19	41	60
S644240.A	12	15	24	21	39	60
S634000.A	10	12	17	19	43	62
S002H00.A	8	8	18	19	43	62
S652300.A	8	8	24	19	44	63
S682V00.A	10	14	23	25	40	65
S662100.A	20	8	25	16	50	66
S682P00.A	8	14	23	26	41	67
S687C00.A	8	11	31	21	46	67
S644120.A	16	14	21	15	52	67
S686001.A	8	12	12	13	58	71
S684300.A	10	12	23	23	49	72
S002300.A	10	9	22	23	50	73
S651300.A	20	8	23	17	56	73
S684500.A	10	12	24	23	51	74
S653300.A	10	12	29	23	52	75
S002A00.A	10	9	20	30	45	75
S652100.A	8	10	26	24	51	75
S687300.A	14	10	19	21	55	76
S687400.A	14	10	19	21	55	76
S686200.A	14	9	21	21	56	77
S686300.A	14	9	21	21	56	77
S671001.A	6	16	16	17	63	80
S652600.A	10	10	27	24	56	80
S653600.A	10	11	29	24	58	82
S682N00.A	8	18	23	33	50	83
S682R00.A	8	18	23	33	50	83
S651200.A	22	9	23	20	65	85
S653100.A	10	13	28	29	57	86
S687800.A	16	10	19	23	66	89
S687G00.A	8	13	26	32	58	90
S684001.A	6	18	17	19	73	92
S002900.A	12	15	23	32	62	94
S686400.A	16	10	24	25	69	94

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	LENGTH N
S684100.A	10	15	21	34	60	94
S682T00.A	8	13	25	38	59	97
S001001.A	8	20	21	21	78	99
S682L00.A	16	15	22	33	66	99
S661400.A	12	16	31	32	70	102
S661320.A	12	19	24	34	71	105
S001100.A	22	15	23	32	75	107
S687500.A	16	13	21	31	78	109
S687600.A	18	11	20	28	82	110
S002600.A	12	14	26	39	71	110
S644170.A	10	9	20	49	61	110
S686100.A	18	13	27	35	81	116
S661001.A	8	23	19	24	93	117
S681500.A	6	12	22	47	70	117
S682E00.A	14	19	31	44	74	118
S671600.A	16	16	27	40	79	119
S687900.A	20	16	23	34	85	119
S622001.A	8	20	23	42	79	121
S682M00.A	18	20	23	45	76	121
S661520.A	14	18	32	44	80	124
S644230.A	16	21	26	47	79	126
S002700.A	14	18	21	49	77	126
S653001.A	6	25	19	26	100	126
S001600.A	32	16	26	36	91	127
S686500.A	20	15	25	36	92	128
S652001.A	6	25	21	26	103	129
S644100.A	8	27	23	28	105	133
S661530.A	14	14	28	46	93	139
S686800.A	24	17	23	43	100	143
S687001.A	6	29	19	30	116	146
S651100.A	32	16	32	39	108	147
S644150.A	16	14	20	67	83	150
S661500.A	14	31	21	32	125	157
S681600.A	6	12	23	68	91	159
S686700.A	24	19	24	49	111	160
S632000.A	12	39	26	46	115	161
S672000.A	14	34	19	36	126	162
S687700.A	26	16	21	47	127	174
S631000.A	14	42	26	49	125	174
S634001.A	36	16	20	53	122	175
S686600.A	24	21	24	55	122	177
S602000.A	14	40	23	42	138	180
S682C00.A	18	26	35	71	109	180
S682U00.A	12	27	38	72	110	182
S682X00.A	12	35	30	61	122	183
S661300.A	14	35	27	52	131	183

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	LENGTH N
S644220.A	20	31	31	63	120	183
S611000.A	10	34	21	62	126	188
S682B00.A	18	27	36	76	114	190
S612000.A	10	34	21	62	129	191
S652500.A	22	29	37	70	122	192
S661310.A	14	27	27	65	130	195
S644210.A	12	18	25	79	117	196
S653500.A	24	30	37	72	124	196
S682Y00.A	14	35	31	66	136	202
S682G00.A	30	25	25	70	133	203
S682900.A	18	24	31	74	129	203
S684400.A	18	40	39	70	139	209
S644200.A	24	36	37	72	142	214
S652400.A	20	26	31	82	135	217
S644160.A	16	13	21	101	117	218
S644110.A	18	17	29	92	134	226
S661510.A	20	28	38	80	147	227
S653400.A	22	27	31	88	141	229
S682700.A	22	24	33	91	138	229
S682001.A	18	48	23	49	188	237
S686900.A	26	24	27	80	157	237
S614000.A	12	45	15	81	158	239
S682F00.A	18	29	36	106	148	254
S682A00.A	20	32	38	110	154	264
S682D00.A	20	32	38	110	154	264
S644130.A	16	14	25	118	147	265
S644180.A	16	19	23	123	163	286
S644122.A	22	18	27	134	175	309
S651000.A	30	51	24	65	253	318
S686A00.A	44	44	48	108	218	326
S632001.A	6	48	38	116	229	345
S002001.A	14	74	25	75	273	348
S681400.A	12	17	26	151	198	349
S631001.A	6	50	38	120	241	361
S682H00.A	38	45	38	133	236	369
S686B00.A	34	48	50	127	245	372
S662300.A	52	58	46	119	260	379
S661900.A	36	48	35	131	250	381
S683000.A	22	65	30	84	297	381
S682100.A	38	33	36	141	243	384
S671000.A	42	63	23	71	321	392
S682800.A	24	32	36	173	232	405
S644140.A	14	14	26	182	226	408
S602001.A	14	52	38	132	292	424
S681700.A	8	12	23	206	229	435
S687200.A	24	34	37	177	276	453

CAMP FILENAME	FILE SIZE (K)	n2	n1	N2	N1	LENGTH N
S672001.A	56	61	37	146	313	459
S687100.A	28	47	43	181	310	491
S623000.A	22	80	48	136	375	511
S688000.A	80	85	36	106	423	529
S684000.A	46	89	31	104	440	544
S623001.A	12	52	52	146	423	569
S682600.A	80	71	47	195	397	592
S682J00.A	44	45	40	258	406	664
S001000.A	64	111	36	126	541	667
S687H00.A	16	41	44	287	447	734
S681000.A	60	116	44	135	607	742
S681200.A	20	29	28	343	450	793
S662000.A	66	156	38	171	683	854
S686000.A	84	137	27	179	688	867
S682200.A	58	60	42	357	557	914
S682K00.A	52	64	40	381	547	928
S652000.A	60	148	34	170	792	962
S682300.A	54	49	44	402	589	991
S653000.A	70	180	34	217	960	1177
S622000.A	46	149	61	362	855	1217
S682500.A	68	73	47	500	732	1232
S687000.A	92	206	46	253	993	1246
S661000.A	86	224	43	262	1056	1318
S683001.A	78	152	79	431	974	1405
S621000.A	26	207	47	416	1002	1418
S621001.A	26	196	70	672	1615	2287
	19	30	26	73	163	236
	307	1522	114	9393	51783	100486
	18	39	11	97	228	317

VOCAB n	VOLUME V	POTEN VOL V* (L^)	RECALIBRATED COMPILE TIME PREDICTION MODEL (T)	TIME ACTUAL (USER)
15	70.32	0.125	2.59	1.2
15	70.32	0.125	2.59	1.2
10	63.12	0.214	2.89	0.8
18	91.74	0.080	2.47	0.8
18	91.74	0.080	2.47	1.5
14	87.57	0.214	3.16	1.6
11	83.03	0.229	3.17	2.2
11	83.03	0.229	3.17	1.5
20	103.73	0.083	2.58	2.5
19	110.45	0.076	2.57	1.5
19	118.94	0.076	2.62	1.7
20	121.01	0.071	2.59	0.9
20	121.01	0.071	2.59	0.9
24	142.13	0.074	2.73	2.5
20	133.98	0.074	2.68	2.6
22	142.70	0.083	2.81	0.9
20	138.30	0.056	2.51	1.6
23	144.75	0.078	2.78	1.4
20	138.30	0.056	2.51	0.9
21	144.95	0.069	2.70	2.5
21	144.95	0.104	3.00	1.6
22	151.62	0.083	2.86	1.0
24	160.47	0.075	2.82	2.6
24	165.06	0.082	2.92	0.9
24	178.81	0.056	2.69	1.3
21	175.69	0.052	2.63	1.4
25	185.75	0.065	2.83	0.9
26	188.02	0.068	2.88	2.1
21	184.48	0.222	3.89	1.7
28	201.91	0.062	2.85	2.6
29	213.75	0.059	2.86	2.7
23	208.08	0.063	2.89	0.9
23	208.08	0.063	2.89	1.0
30	225.72	0.048	2.76	1.4
28	230.75	0.057	2.90	2.8
28	230.75	0.057	2.90	2.6
26	225.62	0.043	2.68	1.0
25	222.91	0.121	3.50	2.7
22	222.97	0.118	3.47	2.2
31	247.71	0.051	2.87	1.9
26	235.02	0.040	2.66	1.0
27	242.50	0.047	2.79	0.9
27	242.50	0.047	2.79	1.0
29	247.76	0.055	2.93	1.4
31	272.48	0.036	2.70	2.0

VOCAB n	VOLUME V	POTEN VOL V* (L^)	RECALIBRATED COMPILE TIME PREDICTION MODEL (T)	TIME ACTUAL (USER)
30	269.88	0.050	2.93	2.8
32	280.00	0.062	3.12	1.2
26	267.93	0.049	2.90	1.4
28	274.02	0.182	4.11	2.0
25	264.70	0.055	2.99	2.6
33	297.62	0.044	2.90	2.2
30	294.41	0.045	2.91	0.9
39	317.12	0.060	3.19	1.1
29	301.19	0.074	3.34	1.5
26	291.43	0.047	2.93	1.0
32	315.00	0.035	2.78	1.9
37	338.61	0.049	3.08	1.4
33	332.93	0.040	2.92	2.9
37	349.03	0.047	3.08	1.3
42	361.29	0.034	2.85	1.5
35	343.66	0.089	3.62	1.1
24	325.53	0.154	4.12	3.7
35	369.31	0.045	3.10	2.8
31	361.66	0.036	2.89	1.0
31	361.66	0.041	3.00	2.1
36	382.57	0.043	3.09	2.9
41	401.82	0.036	2.98	2.4
29	364.35	0.030	2.77	1.0
36	387.74	0.032	2.87	1.9
29	369.21	0.050	3.18	1.5
29	369.21	0.050	3.18	1.6
30	377.83	0.041	3.03	3.3
30	377.83	0.041	3.03	3.2
32	400.00	0.118	4.06	2.6
37	416.76	0.031	2.90	2.1
40	436.40	0.032	2.95	2.4
41	444.68	0.047	3.30	1.3
41	444.68	0.047	3.30	1.3
32	425.00	0.039	3.10	2.0
41	460.75	0.032	3.00	2.5
29	432.36	0.046	3.24	1.7
39	475.69	0.031	3.01	1.4
35	471.89	0.111	4.19	2.8
38	493.31	0.041	3.26	1.0
34	478.22	0.033	3.07	3.6
36	485.97	0.042	3.27	2.8
38	509.05	0.027	2.96	1.3
41	530.40	0.091	4.09	1.7
37	515.74	0.041	3.31	1.5
47	566.57	0.032	3.18	2.9

VOCAB n	VOLUME V	POTEN VOL V* (L^)	RECALIBRATED COMPILE TIME PREDICTION MODEL (T)	TIME ACTUAL (USER)
43	569.76	0.047	3.51	1.3
38	561.53	0.041	3.37	2.2
34	554.53	0.040	3.34	1.7
31	544.96	0.039	3.31	1.7
40	585.41	0.028	3.08	1.1
29	534.38	0.018	2.70	1.0
40	617.34	0.028	3.12	3.4
42	630.90	0.101	4.41	2.8
34	595.23	0.023	2.96	1.4
50	665.98	0.028	3.20	1.5
43	645.73	0.030	3.22	3.3
39	628.96	0.041	3.48	1.8
43	656.58	0.041	3.53	1.0
43	656.58	0.039	3.47	1.5
50	699.84	0.026	3.17	3.6
47	699.88	0.034	3.42	1.2
39	665.96	0.035	3.39	1.1
44	687.89	0.101	4.52	2.4
42	684.82	0.034	3.40	2.6
40	681.21	0.033	3.37	3.8
46	712.54	0.092	4.44	2.1
50	750.63	0.084	4.40	1.0
42	749.53	0.022	3.09	3.0
40	761.04	0.034	3.50	3.8
48	815.40	0.102	4.73	4.2
48	820.99	0.026	3.31	2.6
34	763.12	0.021	3.08	1.3
52	894.97	0.092	4.73	3.0
35	815.56	0.015	2.89	1.3
43	868.20	0.032	3.57	4.2
65	969.60	0.065	4.41	1.5
53	927.92	0.099	4.87	3.0
37	906.44	0.032	3.61	2.0
68	1059.22	0.066	4.53	1.7
36	904.74	0.030	3.54	2.8
45	972.06	0.032	3.66	4.4
63	1075.91	0.083	4.83	2.1
61	1067.53	0.021	3.37	1.6
65	1096.07	0.020	3.34	1.6
65	1102.09	0.038	3.98	1.4
62	1089.62	0.050	4.25	2.5
62	1089.62	0.032	3.78	1.3
55	1086.90	0.052	4.30	1.5
63	1135.68	0.020	3.37	1.7
55	1104.24	0.052	4.31	1.5

VOCAB n	VOLUME V	POTEN VOL V* (L^)	RECALIBRATED COMPILE TIME PREDICTION MODEL (T)	TIME ACTUAL (USER)
66	1160.52	0.022	3.51	3.7
54	1122.20	0.031	3.77	1.5
43	1063.55	0.018	3.25	1.2
67	1188.95	0.023	3.53	4.2
66	1220.97	0.034	3.97	1.5
50	1145.70	0.029	3.72	1.9
55	1173.62	0.021	3.45	1.7
79	1317.49	0.029	3.89	4.8
73	1324.62	0.027	3.81	1.4
57	1265.74	0.020	3.50	3.9
34	1109.07	0.012	2.96	1.3
46	1248.33	0.013	3.09	1.3
66	1372.08	0.018	3.48	4.7
58	1341.48	0.020	3.53	4.6
57	1335.73	0.016	3.33	1.7
71	1457.49	0.085	5.28	1.7
51	1344.36	0.022	3.64	4.5
60	1411.75	0.074	5.05	2.2
65	1529.68	0.015	3.41	1.7
70	1618.13	0.015	3.47	1.9
70	1618.13	0.015	3.47	1.7
39	1400.63	0.009	2.95	1.4
42	1542.20	0.013	3.31	1.3
45	1696.98	0.010	3.14	1.5
75	1980.76	0.065	5.35	2.9
92	2126.68	0.017	3.84	4.7
86	2217.06	0.022	4.14	1.1
99	2307.02	0.079	5.86	1.1
43	1893.77	0.009	3.12	1.7
88	2331.86	0.022	4.20	1.0
83	2352.39	0.018	3.99	2.3
98	2460.67	0.015	3.87	4.7
104	2539.47	0.021	4.26	5.2
83	2428.89	0.021	4.20	3.7
95	2503.12	0.052	5.36	2.2
69	2345.67	0.013	3.67	2.3
86	2519.10	0.077	5.96	4.5
68	2465.42	0.010	3.50	1.9
40	2171.35	0.006	2.93	1.4
90	2752.55	0.021	4.33	2.9
35	2231.24	0.005	2.83	1.6
71	2785.84	0.010	3.63	2.2
98	3036.15	0.023	4.55	4.4
90	3187.50	0.012	3.91	2.3
128	3577.00	0.025	4.85	2.5

VOCAB n	VOLUME V	POTEN VOL V* (L^)	RECALIBRATED COMPILE TIME PREDICTION MODEL (T)	TIME ACTUAL (USER)
121	3660.08	0.045	5.71	3.6
120	3757.35	0.055	6.08	5.3
104	3812.55	0.014	4.24	1.5
118	4074.53	0.015	4.46	3.8
85	4255.84	0.009	3.88	2.8
147	4802.18	0.049	6.29	5.0
85	4704.49	0.006	3.69	2.1
160	5432.87	0.039	6.13	6.4
57	4625.48	0.006	3.61	2.3
194	6490.33	0.048	6.79	6.5
164	6379.00	0.057	7.06	7.8
102	6098.60	0.008	4.18	3.2
104	6218.01	0.008	4.26	3.0
182	7222.50	0.051	7.10	5.7
93	6480.31	0.006	3.86	3.2
214	9111.71	0.049	7.47	6.5
210	9388.24	0.013	5.38	4.0
120	8509.29	0.006	4.28	3.7
252	9939.69	0.035	7.03	9.2
267	10624.00	0.040	7.38	7.9
231	11031.71	0.009	5.05	4.3
254	11327.96	0.021	6.37	6.3
266	18422.43	0.008	5.69	2.9
------	---------	------	-------	-------
56	1534.23	0.050	3.63	2.36
2210	5950085.02	0.002	1.03	2.04
47	2439.28	0.040	1.02	1.43

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
1.0	2.2	0.39
1.1	2.3	0.29
1.0	1.8	1.09
1.1	1.9	0.57
1.6	3.1	-0.63
0.9	2.5	0.66
2.5	4.7	-1.53
2.0	3.5	-0.33
1.7	4.2	-1.62
1.6	3.1	-0.53
2.2	3.9	-1.28
0.9	1.8	0.79
0.9	1.8	0.79
1.9	4.4	-1.67
1.8	4.4	-1.72
1.0	1.9	0.91
1.7	3.3	-0.79
1.7	3.1	-0.32
0.9	1.8	0.71
1.9	4.4	-1.70
2.1	3.7	-0.70
0.9	1.9	0.96
1.8	4.4	-1.58
0.9	1.8	1.12
1.2	2.5	0.19
1.2	2.6	0.03
0.9	1.8	1.03
1.8	3.9	-1.02
2.2	3.9	-0.01
1.9	4.5	-1.65
1.9	4.6	-1.74
0.9	1.8	1.09
0.9	1.9	0.99
1.2	2.6	0.16
1.9	4.7	-1.80
2.0	4.6	-1.70
1.0	2.0	0.68
1.9	4.6	-1.10
1.2	3.4	0.07
1.9	3.8	-0.93
1.0	2.0	0.66
1.0	1.9	0.89
1.0	2.0	0.79
1.1	2.5	0.43
1.8	3.8	-1.10
1.8	4.6	-1.67

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
1.0	2.2	0.92
1.2	2.6	0.30
2.2	4.2	-0.09
2.0	4.6	-1.61
2.2	4.4	-1.50
0.9	1.8	1.11
1.0	2.1	1.09
1.6	3.1	0.24
0.9	1.9	1.03
1.8	3.7	-0.92
1.1	2.5	0.58
2.0	4.9	-1.98
1.1	2.4	0.68
1.1	2.6	0.25
1.0	2.1	1.52
2.6	6.3	-2.18
1.9	4.7	-1.60
0.9	1.9	0.99
1.6	3.7	-0.70
1.9	4.8	-1.71
1.8	4.2	-1.22
1.0	2.0	0.77
1.8	3.7	-0.83
1.1	2.6	0.58
1.2	2.8	0.38
1.9	5.2	-2.17
2.1	5.3	-2.27
2.5	5.1	-1.04
1.8	3.9	-1.00
1.7	4.1	-1.15
1.1	2.4	0.90
1.0	2.3	1.00
1.6	3.6	-0.50
1.9	4.4	-1.40
1.1	2.8	0.44
1.2	2.6	0.41
2.6	5.4	-1.21
1.0	2.0	1.26
2.0	5.6	-2.53
1.9	4.7	-1.43
1.1	2.4	0.56
2.3	4.0	0.09
1.1	2.6	0.71
1.9	4.8	-1.62
1.3	2.6	0.91
1.7	3.9	-0.53

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
1.2	2.9	0.44
1.2	2.9	0.41
1.0	2.1	0.98
0.9	1.9	0.80
2.0	5.4	-2.28
2.5	5.3	-0.89
1.2	2.6	0.36
1.1	2.6	0.60
1.9	5.2	-1.98
1.2	3.0	0.48
1.1	2.1	1.43
1.1	2.6	0.87
2.0	5.6	-2.43
1.0	2.2	1.22
1.1	2.2	1.19
2.5	4.9	-0.38
1.6	4.2	-0.80
2.0	5.8	-2.43
2.4	4.5	-0.06
0.9	1.9	2.50
2.0	5.0	-1.91
2.1	5.9	-2.40
1.4	5.6	-0.87
1.8	4.4	-1.09
1.0	2.3	0.78
2.1	5.1	-0.37
1.2	2.5	0.39
2.0	6.2	-2.63
1.2	2.7	1.71
2.5	5.5	-0.63
1.4	3.4	0.21
1.1	2.8	1.73
1.8	4.6	-1.06
2.0	6.4	-2.74
2.2	4.3	0.53
1.1	2.7	0.67
1.1	2.7	0.64
1.0	2.4	1.58
1.6	4.1	0.15
1.0	2.3	1.48
1.0	2.5	1.80
1.1	2.8	0.57
1.1	2.6	1.71
2.2	5.9	-2.39
1.3	2.8	0.97
0.9	2.1	1.15

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
2.1	6.3	-2.77
1.1	2.6	1.37
1.2	3.1	0.62
1.0	2.7	0.75
2.2	7.0	-3.11
1.0	2.4	1.41
2.1	6.0	-2.50
1.1	2.4	0.56
1.0	2.3	0.79
2.1	6.8	-3.32
1.9	6.5	-2.97
1.1	2.8	0.53
1.0	2.7	2.58
1.8	6.3	-2.66
1.1	3.3	1.75
1.0	2.7	0.71
1.1	3.0	0.47
1.2	2.9	0.57
1.0	2.4	0.55
1.0	2.3	1.01
1.1	2.6	0.54
2.2	5.1	0.25
1.5	6.2	-2.36
0.8	1.9	2.24
1.0	2.1	3.76
1.2	2.9	0.22
1.0	2.0	2.20
1.0	3.3	0.69
1.5	6.2	-2.33
1.8	7.0	-2.74
1.5	5.2	-1.00
1.1	3.3	2.06
1.1	3.4	0.27
2.7	7.2	-1.24
1.1	3.0	0.50
1.0	2.4	0.53
2.4	5.3	-0.97
1.1	2.7	0.13
1.2	3.4	0.23
1.8	6.2	-1.65
1.3	3.6	0.31
1.1	3.6	1.25
1.2	4.8	0.91
2.7	8.0	-1.92
0.9	2.4	1.84
1.3	5.1	-0.64

TIME ACTUAL (SYSTEM)	ACTUAL COMPILE TIME	DIFF PREDICTED - ACTUAL
1.3	4.1	-0.22
2.7	7.7	-1.41
1.2	3.3	0.39
3.0	9.4	-3.27
1.2	3.5	0.11
3.0	9.5	-2.71
3.0	10.8	-3.74
1.2	4.4	-0.22
1.3	4.3	-0.04
2.7	8.4	-1.30
1.2	4.4	-0.54
2.8	9.3	-1.83
1.5	5.5	-0.12
1.2	4.9	-0.62
3.0	12.2	-5.17
2.9	10.8	-3.42
1.3	5.6	-0.55
1.4	7.7	-1.33
1.1	4.0	1.69
-------	-------	-------
1.52	3.88	-0.25
0.31	3.50	2.03
0.55	1.87	1.43

NEW STATISTICS FOR COMPILE TIME MODEL
(FROM BOTTOM OF LAST COLUMN)

MEAN ERROR = - .25

VARIANCE = 2.03

STDRD DEV = 1.43

Appendix B: User's Manual for the SCA

Target SCA Environment

The SCA was developed in a UNIX environment on a VAX 11/785 minicomputer. The SCA is written in the C programming language, hence the target environment must support a C language compiler with the standard library routines and "header" files available (e.g., ctype.h, stdio.h, math.h). All input/output operations use standard C function calls. The SCA always uses standard file input/output for the input Ada source module and all printed output, including the compilation predictions.

Porting the SCA to a New Environment

To port the SCA to a new environment, the software development tools LEX and YACC must also be available in the target environment, and the allocation of data structures and memory space provided for LEX and YACC must be sufficient for the size of the Ada.l and Ada.y input files. To use the SCA in a new hardware or software environment, the SCA should first be installed according to the following procedure:

1. Load the files Ada.l, Ada.y, and y.tab.h in a directory on the target machine.

2. In that directory, type the command "Lex Ada.l". This will produce the file "lex.yy.c".

3. Next, type the command "yacc ada.y". Ada.y "includes" lex.yy.c so the latter must

exist prior to this step. The file "y.tab.c" is the output from the YACC program.

4. Now the file y.tab.c must be compiled with the C language compiler. To do so, type the command and parameters:

```
cc y.tab.c -ll -ly -lm
```

This will invoke the C compiler on UNIX-based machines. The parameters inform the linker to look in particular C language libraries to resolve external references (function calls).

5. Step 4 produces the executable SCA which, in a UNIX-based environment, is the file "a.out". You can rename "a.out" to "sca" with the "mv" command if desired.

6. To predict a range of compilation times for a given Ada input module, proceed as follows:

  a. Load the file(s) containing the Ada input module(s) into the directory containing a.out (or whatever you named it). We recommend the Ada "package" as the ideal unit to analyze with the SCA, although any compilation unit will work just fine.

  b. Using the input/output redirection facility of your target machine, invoke the executable SCA file from the command line and redirect the input to be one Ada source file. It is a good idea to also redirect the output to a file, but it is not necessary. You can name the output file whatever you desire.

  c. If output is not redirected to a file, the SCA will echo each line of the Ada source file to the screen. When the end of the Ada file is reached, the SCA will scroll the screen with data pertaining to the counting of the Software Science measures. You may stop the screen and inspect this information if you wish. Eventually the results of the compilation timing model computations will appear on the screen. The range of compilation times will be shown for each of four distinct computer environments. Copy the appropriate information for later reference.

d. If output is redirected to a file, the same information as in step "c" above is saved to a file. To quickly obtain the timing prediction, open this file (use any editor) and scroll to the end. The predicted compilation ranges are shown there just as in step "c" above.

e. Example for a UNIX-based environment:

```
SCA <source.ada >source.out
```

Source.ada is the Ada source file. The "<" redirects standard input from the keyboard to the file "source.ada". Similarly, "<" redirects output from the terminal to the file "source.out". Source.out contains all the information which the SCA produces with respect to the Ada input file, including the compilation range predictions.

Implementing Other Models

There are other Software Science measures which may be computed based on n_1, n_2, N_1, and N_2. Indeed, any other metrics which are derived from these elementary measures could also be implemented in the SCA. Table 2, Halstead's Equations, show a few immediate candidates. However, regardless of which new model is considered for implementation in the SCA, it must adopt the token counting strategy rules of Chapter II as a given. These rules, for all intents and purposes, are fixed in the SCA. Modification of the token counting rules requires an intimate understanding of the development of the SCA and should not be attempted without careful consideration.

Nevertheless, it is easy to augment the SCA with a new metric. As mentioned above, the metric must be derivable from n_1, n_2, N_1, and N_2, and must adopt the SCA's token counting strategy. The following procedures will integrate a new complexity metric with the SCA:

1. Implement the new metric as a function call in the C language. There are no parameters to pass, and the function may return an integer (default) value only. Remember that n_1, n_2, N_1, and N_2 are global variables and can be accessed by any function in the SCA. Print your results to standard output (i.e., printf() function is fine).

2. Insert the code for the function call just after the function "calc_metrics()" in the file "y.tab.c". Place a call to your new function as the last line in the function "main()" also in y.tab.c.

3. Re-compile y.tab.c according to the instructions in the last section. Any library calls the new function uses (apart from the libraries already referenced during linking) must be added to the command line to compile y.tab.c.

4. Run the SCA as usual. Your results will appear as the last printed output on the terminal screen or in the output file. Don't forget to document your modifications and additions to the SCA.

The ADA Input Modules and Errors

The SCA can only process one Ada input source file at a time. The source code should be syntactically correct with respect to the Ada Language Reference Manual (LRM). Some syntax errors cause the SCA to abort, and some do not. The facility embedded in the SCA to recover from syntax errors is crude at best. If the SCA aborts, the program just stops, all open files are closed, and the SCA output stream ends where the error occurred. On the other hand, if the syntax error is minor, the SCA may be able to recover. In that case, the SCA will issue a syntax error message to the UNIX-environment specific file, "stderror", before processing continues. Stderror defaults to your terminal screen. Our experience shows that the compilation timing predictions are not adversely effected by minor, recoverable syntax errors. However, errors which cause the SCA to abort must, of course, be corrected before any results at all can be obtained from the file containing the error. In this case you must delete the original output file (if redirection is used), correct the syntax error, and re-run the SCA.

Appendix C: Sample SCA Output

nrpca1.o

```
Starting SCA scanning of input Ada file

with INSTRUMENT;
[2] use INSTRUMENT;
[3]
[4] procedure NRPCA1 is
[5]
[6] package PS_CS is new PROCS(BOOLEAN);
[7] use PS_CS;
[8] package B is new PROCS(BOOLEAN);
[9]   TTRUE : B.T := B.T(TRUE);
[10]   TFALSE : B.T := B.T(FALSE);
[11]   TEST : B.T := B.Ident(TFALSE);
[12]   Recursion : B.T := B.T(test);
[13]   procedure Nested_Recursive_Procedure is
[14]
[15]    Local_1 : T;
[16]    Local_2 : T;
[17]
[18]    procedure Nested is
[19]     begin  --Nested
[20]       if BOOLEAN(Recursion) then
[21]         B.Let(Recursion, B.Ident(TFALSE));
[22]         Nested_Recursive_Procedure;
[23]       else
[24]         B.Let(Test, B.Ident(TFALSE));
[25]         if BOOLEAN(Test) then
[26]           Nested_Recursive_Procedure;
[27]         end if;
[28]      end if;
[29]    end Nested;
[30]
[31]    begin --Nested_Recursive_Procedure
[32]      if BOOLEAN(Recursion) then
[33]        Let(Local_1, Ident(Init));
[34]        Nested;
[35]      elsif not BOOLEAN(Test) then
[36]        Let(Local_2, Ident(Init));
[37]        Nested;
[38]      end if;
[39]  end Nested_Recursive_Procedure;
[40]
[41] begin --NRPCA1
[42] START("NRPCA1","Nested Recursion Procedure Call (Control)");

[43] for I in 1..100000 loop
```

```
   B.let(recursion, B.ident(test));
   Nested_Recursive_Procedure;
end loop;
STOP;

end NRPCA1;
---> SCA analysis complete: 0 syntax error(s) <---

THIS IS THE LIST OF DELIMITER OPERATORS FOR THE MODULE SCANNED

There were 0 * operators in the module scanned
There were 0 + operators in the module scanned
There were 0 - operators in the module scanned
There were 0 / operators in the module scanned
There were 0 & operators in the module scanned
There were 0 add unary operators
There were 0 minus unary operators
There were 30 ; operators in the module scanned
There were 0 > operators in the module scanned
There are 0 >= operators in the module scanned
There were 0 < operators in the module scanned
There were 0 <= operators in the module scanned
There were 0 = operators in the module scanned
There were 0 /= operators in the module scanned
There were 0 attribute apostrophes
There were 0 aggregate apostrophes
There were 0 ¦   operators in the module
There were 0 ' ' operators in the module
There were 0 # # operators in the module
There were 2 " " operators in module
There were 1 ..  operators in the module
There were 0 **  operators in the module
There are 4  :=  operators in the module
There were 0 =>  operators in the module
There were 0 < > operators in the module
There were 0 <<>> operators in the module
There were 14  .  operators in the module
There were 6  :  operators in the module
There were 6  ,  operators in the module
There are 0 declaration parentheses
There are 12 invocation parentheses
Also 0 dimensioning parentheses
There are 0 subscript parentheses
There are 2 aggregate parentheses
There are 0 enumeration parentheses
There are 0 expression parentheses
Also 7 type convert parentheses
Also 0 dimensioning brackets
There are 0 subscript brackets
The number of do end operators is 0
The number of body is operators is 0
The number of or else operators is 0
The number of goto operators is 0
The number of for use operators is 0
The number of and then operators is 0
The number of array of operators is 0
The number of subtype is operators is 0
The number of elsif then operators is 1
The number of loop end loop operators is 0
The number of limited private operators is 0
The number of if then end if operators is 3
The number of begin end operators is 3
Number of record end record operators: 0
Number of exception when operators: 0
Number of select end select operators: 0
Number of function return operators: 0
Number of for in loop end loop operators: 1
Number of case is when end case operators: 0
Number of declare begin end operators: 0
Number of while loop end loop operators: 0
Number of at single-word operators: 0
Number of when single-word operators: 0
Number of use single-word operators: 2
Number of and single-word operators: 0
Number of mod single-word operators: 0
Number of end single-word operators: 0
Number of not single-word operators: 1
Number of all single-word operators: 0
Number of new single-word operators: 0
Number of out single-word operators: 0
Number of rem single-word operators: 0
Number of abs single-word operators: 0
Number of else single-word operators: 1
Number of type single-word operators: 0
Number of task single-word operators: 0
Number of with single-word operators: 1
Number of exit single-word operators: 0
Number of raise single-word operators: 0
Number of abort single-word operators: 0
Number of delta single-word operators: 0
Number of entry single-word operators: 0
Number of accept single-word operators: 0
Number of delay single-word operators: 0
Number of range single-word operators: 0
Number of others single-word operators: 0
Number of digits single-word operators: 0
Number of return single-word operators: 0
Number of access single-word operators: 0
Number of generic single-word operators: 0
Number of private (declaration) single-word operators: 0
Number of pragma single-word operators: 0
Number of reverse single-word operators: 0
Number of renames single-word operators: 0
Number of constant single-word operators: 0
Number of separate (detail) single-word operators: 0
Number of package single-word operators: 0
Number of procedure single-word operators: 3
Number of terminate single-word operators: 0
Number of exception (declaration) single-word operators: 0
Number of separate (declaration) single-word operators: 0
Number of private (detail) single-word operators: 0
Number of or boolean single-word operators: 0
Number of or alternative single-word operators: 0
Number of is single-word operators: 3
Number of in mode single_word operators: 0
Number of in membership single_word operators: 0
Number of xor single_word operators: 0
Number of generic instantiation operators for packages: 2
Number of generic instantiation operators for functions: 0
Number of generic instantiation operators for procedures: 0


                THE SCA IDENTIFIER TABLE FOR THE INPUT FILE

IDENTIFIER              TYPE                       TIMES USED AS
                                                OPERAND        OPERATOR

 BOOLEAN                  type                  0              2
 INTEGER                  type                  0              0
 FLOAT                    type                  0              0
 STRING                   type                  0              0
 NATURAL                  type                  0              0
 POSITIVE                 type                  0              0
 DURATION                 type                  0              0
 INSTRUMENT               var/const             2              0
 NRPCA1                   subprgram             2              0
 PS_CS                    var/const             2              0
 PROCS                    var/const             2              0
 B                        var/const            15              0
 TTRUE                    var/const             1              0
 T                        type                  0              6
 T                        type_conv             0              3
 TRUE                     var/const             1              0
 TFALSE                   var/const             4              0
 FALSE                    var/const             1              0
 TEST                     var/const             6              0
 IDENT                    unknownop             0              6
 RECURSION                var/const             5              0
 NESTED_RECURSIVE_PROCEDURE   subprgram         2              3
 LOCAL_1                  var/const             2              0
 LOCAL_2                  var/const             2              0
 NESTED                   subprgram             2              2
 BOOLEAN                  type_conv             0              4
 LET                      unknownop             0              5
 INIT                     var/const             2              0
 START                    subprgram             0              1
 "NRPCA1"                  strg_lit          1              0
 "NESTED RECURSION PROCEDURE CALL (CONTROL)" strg_lit  1     0
 I                         loop_var          1              0
 1                         numrlitrl         1              0
 100000                    numrlitrl         1              0
 STOP                      subprgram         0              1


THE SOFTWARE SCIENCE MEASURES ARE AS FOLLOWS:
number of distinct operands:  n2 =     21
number of distinct operators: n1 =     30
total number of operands:     N2 =     56
total number of operators:    N1 =    136


A SAMPLE OF HALSTEAD'S SOFTWARE SCIENCE EQUATIONS FOLLOWS
The length N of the module is 192.00
The volume V of the module is 1089.11
The estimated level of implementation of the module is 0.0250
The estimated effort to program the module is 43564.23
Note: You must have a basic knowledge of Software Science
theory to interpret this data.


MILLER'S ORIGINAL COMPILE TIME MODEL PREDICTIONS ARE:

Prediction for UNIX ASC environment is: 4.04 seconds

Prediction for the AOS/VS environment is: 7.55 seconds
Note: last prediction not calibrated for AOS/VS environment
Prediction for the VMS ISL environment is: 4.79 seconds
Note: last prediction not calibrated for VMS ISL environment
Prediction for the VMS CSC environment is: 4.31 seconds
Note: last prediction not calibrated for VMS CSC environment


RESULTS OF UNIX ASC CALIBRATED COMPILE TIME MODEL ARE:

Prediction for UNIX ASC environment is: 3.55 seconds

Prediction for AOS/VS environment is: 3.55 seconds
Note: last prediction not calibrated for AOS/VS environment
Prediction for VMS ISL environment is: 3.55 seconds
Note: last prediction not calibrated for VMS ISL environment
Prediction for VMS CSC environment is: 3.55 seconds
Note: last prediction not calibrated for VMS CSC environment
```

opceal.o

```
Starting SCA scanning of input Ada file

--THIS IS A TEST FILE FOR THE SCA
[2]
[3] with INSTRUMENT;
[4] use INSTRUMENT;
[5]
[6] procedure OPCEA1 is
[7] package NEW_PROCS is new PROCS(INTEGER);
[8] use NEW_PROCS;
[9]   Global_1 : T;
[10]   Global_2 : T;
[11]   Global_3 : T;
[12]   Global_4 : T;
[13]
[14]  function Function_1 (Input : T) return T is
[15]   begin
[16]    if Input = Init then
[17]     return Init/Init;
[18]    end if;
[19]    return Function_1(Init);
[20]  end Function_1;
[21]
[22]  function Function_2 (Input : T) return T is
[23]   begin
[24]     if Input /= Init then
[25]       return Function_2(Init);
[26]     end if;
[27]   return Init/Init;
[28]  end Function_2;
[29]
[30] begin --OPCEA1
[31]   START("OPCEA1","Optimization Perf., Call Elim.(Control)");

[32]    for I in 1..100 loop
[33]      Let (Global_1, Ident(Init));
[34]      Let (Global_2, Ident(Init));
[35]      Let (Global_3, Ident(Init));
[36]      Let (Global_4, Ident(Init));
[37]
[38]      if Ident(Init) = Init then
[39]        Global_1 := T(T(Function_2(Init) * Global_4) /
Function_1(Init));
[40]      else
[41]        Global_2 := T(T(Function_1(Init) * Global_4) /
Function_1(Init));
[42]      end if;
[43]      Let (Global_1, Ident(Init));
[44]      Let (Global_2, Ident(Init));
[45]      Let (Global_3, Ident(Init));
```

```
[46]      Let (Global_4, Ident(Init));
[47]   end loop;
[48]   STOP;
[49] end OPCEA1;
[50] ---> SCA analysis complete: 0 syntax error(s) <---

THIS IS THE LIST OF DELIMITER OPERATORS FOR THE MODULE SCANNED

There were 2 * operators in the module scanned
There were 0 + operators in the module scanned
There were 0 - operators in the module scanned
There were 4 / operators in the module scanned
There were 0 & operators in the module scanned
There were 0 add unary operators
There were 0 minus unary operators
There were 31 ; operators in the module scanned
There were 0 > operators in the module scanned
There are 0 >= operators in the module scanned
There were 0 < operators in the module scanned
There were 0 <= operators in the module scanned
There were 2 = operators in the module scanned
There were 1 /= operators in the module scanned
There were 0 attribute apostrophes
There were 0 aggregate apostrophes
There were 0 ¦   operators in the module
There were 0 ' ' operators in the module
There were 0 # # operators in the module
There were 2 " " operators in module
There were 1 ..  operators in the module
There were 0 **  operators in the module
There are 2  :=  operators in the module
There were 0 =>  operators in the module
There were 0 < > operators in the module
There were 0 <<>> operators in the module
There were 0  .  operators in the module
There were 6  :  operators in the module
There were 9  ,  operators in the module
There are 2 declaration parentheses
There are 24 invocation parentheses
Also 0 dimensioning parentheses
There are 0 subscript parentheses
There are 1 aggregate parentheses
There are 0 enumeration parentheses
There are 0 expression parentheses
Also 4 type convert parentheses
Also 0 dimensioning brackets
There are 0 subscript brackets
The number of do end operators is 0
The number of body is operators is 0
The number of or else operators is 0
The number of goto operators is 0
The number of for use operators is 0
The number of and then operators is 0
The number of array of operators is 0
The number of subtype is operators is 0
The number of elsif then operators is 0
The number of loop end loop operators is 0
The number of limited private operators is 0
The number of if then end if operators is 3
The number of begin end operators is 3
Number of record end record operators: 0
Number of exception when operators: 0
Number of select end select operators: 0
Number of function return operators: 2
Number of for in loop end loop operators: 1
Number of case is when end case operators: 0
Number of declare begin end operators: 0
Number of while loop end loop operators: 0
Number of at single-word operators: 0
Number of when single-word operators: 0
Number of use single-word operators: 2
Number of and single-word operators: 0
Number of mod single-word operators: 0
Number of end single-word operators: 0
Number of not single-word operators: 0
Number of all single-word operators: 0
Number of new single-word operators: 0
Number of out single-word operators: 0
Number of rem single-word operators: 0
Number of abs single-word operators: 0
Number of else single-word operators: 1
Number of type single-word operators: 0
Number of task single-word operators: 0
Number of with single-word operators: 1
Number of exit single-word operators: 0
Number of raise single-word operators: 0
Number of abort single-word operators: 0
Number of delta single-word operators: 0
Number of entry single-word operators: 0
Number of accept single-word operators: 0
Number of delay single-word operators: 0
Number of range single-word operators: 0
Number of others single-word operators: 0
Number of digits single-word operators: 0
Number of return single-word operators: 4
Number of access single-word operators: 0
Number of generic single-word operators: 0
Number of private (declaration) single-word operators: 0
Number of pragma single-word operators: 0
Number of reverse single-word operators: 0
Number of renames single-word operators: 0
Number of constant single-word operators: 0
Number of separate (detail) single-word operators: 0
Number of package single-word operators: 0
Number of procedure single-word operators: 1
Number of terminate single-word operators: 0
Number of exception (declaration) single-word operators: 0
Number of separate (declaration) single-word operators: 0
Number of private (detail) single-word operators: 0
Number of or boolean single-word operators: 0
Number of or alternative single-word operators: 0
Number of is single-word operators: 3
Number of in mode single_word operators: 0
Number of in membership single_word operators: 0
Number of xor single_word operators: 0
Number of generic instantiation operators for packages: 1
Number of generic instantiation operators for functions: 0
Number of generic instantiation operators for procedures: 0


          THE SCA IDENTIFIER TABLE FOR THE INPUT FILE

IDENTIFIER          TYPE                 TIMES USED AS
                                        OPERAND      OPERATOR

 BOOLEAN                 type            0              0
 INTEGER                 type            0              1
 FLOAT                   type            0              0
 STRING                  type            0              0
 NATURAL                 type            0              0
 POSITIVE                type            0              0
 DURATION                type            0              0
 INSTRUMENT              var/const       2              0
 OPCEA1                  subprgram       2              0
 NEW_PROCS               var/const       2              0
 PROCS                   var/const       1              0
 GLOBAL_1                var/const       4              0
 T                       type            0              8
 GLOBAL_2                var/const       4              0
 GLOBAL_3                var/const       3              0
 GLOBAL_4                var/const       5              0
 FUNCTION_1              subprgram       2              4
 INPUT                   var/const       3              0
 INIT                    var/const      22              0
 FUNCTION_2              subprgram       2              2
 INPUT                   var/const       1              0
 START                   subprgram       0              1
 "OPCEA1"                strg_lit        1              0
 "OPTIMIZATION PERF., CALL ELIM.(CONTROL)" strg_lit  1  0
 I                       loop_var        1              0
 1                       numrlitrl       1              0
 100                     numrlitrl       1              0
 LET                     subprgram       0              8
 IDENT                   subprgram       0              9
 T                       type_conv       0              4
 STOP                    subprgram       0              1
```

```
THE SOFTWARE SCIENCE MEASURES ARE AS FOLLOWS:
number of distinct operands:  n2 =     18
number of distinct operators: nl =     33
total number of operands:     N2 =     58
total number of operators:    N1 =    149


A SAMPLE OF HALSTEAD'S SOFTWARE SCIENCE EQUATIONS FOLLOWS
The length N of the module is 207.00
The volume V of the module is 1174.19
The estimated level of implementation of the module is 0.0188
The estimated effort to program the module is 62427.88
Note: You must have a basic knowledge of Software Science
theory to interpret this data.


MILLER'S ORIGINAL COMPILE TIME MODEL PREDICTIONS ARE:

Prediction for UNIX ASC environment is: 4.17 seconds

Prediction for the AOS/VS environment is: 7.66 seconds
Note: last prediction not calibrated for AOS/VS environment
Prediction for the VMS ISL environment is: 4.87 seconds
Note: last prediction not calibrated for VMS ISL environment
Prediction for the VMS CSC environment is: 4.38 seconds
Note: last prediction not calibrated for VMS CSC environment


RESULTS OF UNIX ASC CALIBRATED COMPILE TIME MODEL ARE:

Prediction for UNIX ASC environment is: 3.36 seconds

Prediction for AOS/VS environment is: 3.36 seconds
Note: last prediction not calibrated for AOS/VS environment
Prediction for VMS ISL environment is: 3.36 seconds
Note: last prediction not calibrated for VMS ISL environment
Prediction for VMS CSC environment is: 3.36 seconds
Note: last prediction not calibrated for VMS CSC environment
```

Bibliography

Booch, Grady. Software Engineering With Ada. Menlo Park CA. The Benjamin/Cummings Publishing Company, Inc., 1987.

Fischer, Charles N. and LeBlanc, Richard J.,Jr. Crafting A Compiler. Menlo Park CA. The Benjamin/Cummings Publishing Company,Inc., 1988.

Halstead, Maurice H. Elements of Software Science. New York. Elsevier North-Holland Inc., 1977.

Howatt, Capt James W. A Quantitative Characterization of Control Flow Context: Software Measures for Programming Environments. PhD dissertation. Department of Computer Science, Iowa State University, Ames Iowa, 1985.

----, et al., "A Software Science Counting Strategy For The Full Ada Language", ACM SIGPLAN Notices,vol.22, no.5,(May, 1987) pp. 32-41.

----. Assistant Professor of Computer Systems, School of Engineering. Personal interviews. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, January 1988 through February 1988.

Jayaprakash, S. and others. "MEBOW: A Comprehensive Measure of Control Flow Complexity," Proceedings of the IEEE Computer Software and Applications Conference. 238-244. Washington DC. Computer Society Press of the IEEE, 1987.

Kernighan, Brian W. and Pike, Rob. The Unix Programming Environment. Englewood Cliffs New Jersey. Prentice-Hall, Inc., 1984.

Kernighan, Brian W. and Ritche, Dennis M. The C Programming Language. Englewood Cliffs New Jersey. Prentice-Hall, Inc., 1984.

Levitin, A. V. "How To Measure Software Size, and How Not To," Proceedings of the Software and Applications Conference. 314-318. Washington DC: Computer Society Press of the IEEE, 1986.

Maness, Capt Robert S. Validation of a Structural Model of Computer Compile Time. MS Thesis, AFIT/GCS/ENG/86D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

McCabe, Thomas. "A Complexity Measure," IEEE Transactions on Software Engineering. 308-320 (December, 1976).

Mehndiratta, B., and P. S. Grover. "Software Metrics Applied To Ada," Proceedings of the IEEE Computer Applications Conference. 368-372. Washington DC: Computer Society Press of the IEEE, 1987.

Miller, Dennis M. Application of Halstead's Timing Model to Predict Compilation Time of Ada Compilers. MS Thesis, AFIT/GE/ENG/86D-7. School of Engineering, Air Force Institute of Technology, (AU), Wright-Patterson AFB OH, December 1986.

Ramamurthy, B., and A. Melton. "A Synthesis of Software Science Metrics and the Cyclomatic Number," Proceedings of the IEEE Computer Software and Applications Conference. 308-313. Washington DC : Computer Society Press of the IEEE, 1986.

Shaw, Capt Wade H., Assistant Professor of Electrical Engineering, School of Engineering. Personal interviews. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, January 1988 through February 1988.

Strovink, Capt Mark, System Manager For Unix Computer Systems. Schools of Engineering and Logistics. Personal Interview via Electronic Mail. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, October 1988.

User's Manual. Version Description Document For The Missile Software Parts Of The Common Ada Missile Packages (CAMP). Contract Number F08635-86-C-0025. McDonald Douglass Astronautics Company, St. Louis MO, 30 October 1987.

Van Verth, P. B. "A Program Complexity Model That Includes Procedures," Proceedings of the IEEE Computer Software and Applications Conference. 252-258. Washington DC: Computer Society Press of the IEEE, 1987.

VITA

Eric R. Goepper was born [redacted] In 1973 he graduated from high school [redacted] After transferring from a small Maryland college in 1975, he graduated from Virginia Polytechnic Institute and State University in 1978 with a BS in mathematics. Eric entered the Air Force in 1981 and soon graduated from Officer's Training School. He was then assigned to the 6591st Computer Services Squadron at HQ AFSC, Andrews AFB, Maryland. In 1984 he went to Electronics Systems Division (ESD), Hanscom AFB, Massachusetts where he worked as a project officer for a major information systems acquisition. From 1985 until 1987 Capt Goepper traveled worldwide as the ESD focal point for the Military Airlift Command (MAC) Command and Control (C^2) Upgrade Program. In the spring of 1987 he transferred to Wright-Patterson AFB, Ohio to enter the AFIT graduate school program in computers systems.

[redacted]

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distributed unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/88D-7	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG
7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DOD Ada Validation Facility	8b. OFFICE SYMBOL *(If applicable)* ASD/SCEL
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(City, State, and ZIP Code)* Wright-Patterson AFB, OH 45433	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A SOURCE CODE ANALYZER TO PREDICT COMPILATION TIME FOR AVIONICS SOFTWARE USING SOFTWARE SCIENCE MEASURES UNCLASSIFIED

12. PERSONAL AUTHOR(S)
Eric R. Goepper, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1988 December	151

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer Programming, Compilers, Metrics
Computer Programs, Programming Languages

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Chairman: James W. Howatt, Maj, USAF
Assistant Professor of Electrical Engineering and Computer Science

[illegible] for release in [illegible]
[illegible]
12 Jan 1989

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT.] ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)*	22c. OFFICE SYMBOL
Maj James W. Howatt	513-255-6913	AFIT/ENG

UNCLASSIFIED

Block 19 -con-

Abstract:
This thesis describes the construction of an Ada source code analyzer (SCA) which produces values for the Software Science measures n_1, n_2, N_1, and N_2. The measures are used to evaluate a mathematical model designed to predict the compile time of Ada modules. The primary goal of this effort was to provide a software tool to metrics researchers which could automatically compute Software Science measures for Ada modules. A secondary goal was to produce a convenient method for Ada compiler researchers to predict the amount of time consumed during compilation of given avionics software modules.

As the SCA was built, we incorporated the rules of a new Ada token counting strategy designed to yield meaningful results for entire Ada programs, not just executable code. Once satisfied the SCA implemented the rules correctly and produced accurate counts for the Software Science measures, we added the compile time model to the SCA.

To test the validity of the compile time model, over 200 modules were selected at random from among the Common Ada Missile Packages (CAMP) software suite. For each module, both the compile time as predicted by the SCA and the actual compile time using the Verdix compiler were recorded. Finally, the prediction error values (predicted compile time minus actual compile time) were recorded and analyzed.

For the test environment, in 95% of the test cases the SCA initially overestimated compile time with an average prediction error of 4.35 seconds. Since the average actual compile time was only 3.88 seconds, this average error figure was unacceptable. These results led us to recalibrate the model's parameters. When the recalibrated model was tested, the average error fell to -.25 seconds. Also, the curve of the predicted compile time values now fit the curve of the actual values nicely.

UNCLASSIFIED